Classified
SECURITY CLASSIFICATION OF THIS PAGE

(2)

REPORT DOCUMENTATION PAGE

AD-A213 258

1b. RESTRICTIVE MARKINGS

3 DISTRIBUTION / AVAILABILITY OF REPORT

Unlimited

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

TR 89-1040

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
Cornell University		Office of Naval Research

6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code)
Department of Computer Science Upson Hall, Cornell University Ithaca, NY 14853	800 North Quincy Street Arlington, VA 22217-5000

8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
Office of Naval Research		N000014-86-K-0092

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO
800 North Quincy Street Arlington, VA 22217-5000				

11. TITLE (Include Security Classification)

Designing Distributed Services Using Refinement Mappings

12. PERSONAL AUTHOR(S)

Jacob Itzhack Aizikowitz

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM _____ TO _____	September 29, 1989	81

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed services, client/server, refinement mapping, verification, assertional reasoning

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The thesis addresses the design of multiple-server implementations for services in distributed systems--a generalization of the replication management problem. A frequently used correctness criteria for replication management is that clients of a service not be able to distinguish a single-server implementation from one that involves multiple servers. Our approach formalizes this idea. It is based on viewing a single-server implementation of a service as a specification of that service. A multiple-server implementation is considered correct iff it implements this single-server based specification.

We show how program proof outlines can be viewed as specifications and, using refinement mappings, define what it means for one proof outline to implement another. The notion of a structural refinement, which formalizes the relationship between a program and the result of performing step-wise refinement, is defined. When one proof outline is a structural refinement of the other, simplified proof obligations can be used to show that one implements the other. (con't over)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	

22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Fred B. Schneider	(607) 255-9221	

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Finally, the methodology for designing a multiple-server implementation of a service is presented. The methodology is based on structural refinement and on viewing proof outlines as specifications. A designer defines a refinement mapping to express the relationship between the state space of a given single-server implementation of a service and the state space of the desired multiple-server implementation. Using this refinement mapping, a provably correct multiple-server implementation is derived from the single server one. Different refinement mappings as well as different single-server based specifications result in different implementations. Examples illustrate the concepts and the methodology.

DESIGNING DISTRIBUTED SERVICES USING REFINEMENT MAPPINGS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jacob Itzhack Aizikowitz

January 1990

DESIGNING DISTRIBUTED SERVICES USING REFINEMENT MAPPINGS

Jacob Itzhack Aizikowitz, Ph.D.

Cornell University 1990

The thesis addresses the design of multiple-server implementations for services in distributed systems—a generalization of the replication management problem. A frequently used correctness criteria for replication management is that clients of a service not be able to distinguish a single-server implementation from one that involves multiple servers. Our approach formalizes this idea. It is based on viewing a single-server implementation of a service as a specification of that service. A multiple-server implementation is considered correct iff it implements this single-server based specification.

We show how program proof outlines can be viewed as specifications and, using refinement mappings, define what it means for one proof outline to implement another. The notion of a structural refinement, which formalizes the relationship between a program and the result of performing step-wise refinement, is defined. When one proof outline is a structural refinement of the other, simplified proof obligations can be used to show that the one implements the other.

Finally, a methodology for designing a multiple-server implementation of a service is presented. The methodology is based on structural refinement and on

viewing proof outlines as specifications. A designer defines a refinement mapping to express the relationship between the state space of a given single-server implementation of a service and the state space of the desired multiple-server implementation. Using this refinement mapping, a provably correct multiple-server implementation is derived from the single server one. Different refinement mappings as well as different single-server based specifications result in different implementations. Examples illustrate the concepts and the methodology.

Biographical Sketch

Jacob (actually, Yaacov) Aizikowitz was born April 12, 1951 in Hadera, Israel, to the delight of his parents and sister. He graduated from Hadera's high school eighteen years later, and in August 1969 left home to serve in the army; he now holds the rank of Major (Reserve). During the three years of mandatory service, Jacob's focus shifted from physics to computer science, and in October 1972 he started studying computer science in the Technion, Israel Institute of Technology, Haifa. In 1977, he graduated cum laude with a B.Sc. in computer science.

After graduation, he joined Mini Systems Inc. where, under Amir Pnueli and Hagi Lachover, he worked on the design and implementation of an operating system for Scitex—a pioneer in the field of computerized color pre-press. This system became the backbone of Scitex's products and is being used to this day.

While at Mini Systems, he and Nava Hefetz married and moved to Haifa, where Nava was completing her graduate studies at the Technion for a D.Sc. On February 3, 1981 their daughter Tamar was born in Haifa.

A desire to further his knowledge brought Jacob and his family to Cornell University, where he joined the Computer Science Department as a graduate student. On September 8, 1987, Elad was born. On Elad's second birthday, Jacob handed the final draft of his dissertation to his committee; on September 18, 1989

he successfully defended the dissertation.

Jacob was invited by Efraim Arazi, the founder and former president of Scitex, to join Elecronics for Imaging Inc. (EFI)—a new start-up company founded by Mr. Arazi. He accepted the invitation and currently is Director of Research and Development.

In memory of my dear mother Tamar Aizikowitz, née Mordel.

Acknowledgements

I am thankful to my advisor, Fred B. Schneider, for defining goals far beyond my imagination and for helping me explore the unknown. From Fred I have learned the difference between an idea and a result, between a collection of results and a chapter, and between a collection of chapters and a harmonious dissertation. He taught me that technical problems often appear disguised as language problems and that improving the presentation often results in clarification and refinment of technical ideas. If while reading this dissertation one ever stumbles on a sentence, or feels that ideas do not flow smoothly, it is due to my own impatience.

I would like also to thank the other members of my committee, Vithala R. Rao and Sam Toueg who, despite short notice, carefully read the manuscript and provided detailed and significant comments. Discussions that I had with Sam on this work and related research were educational and enlighting.

Before starting my studies at Cornell, I was fortunate to work with many talented people at Mini Systems Inc. I am most thankful to Amir Pnueli, who guided me at Mini and helped me understand operating systems. Amir encouraged me to pursue higher education and was always there to give advice. His explanation of the meaning of the formula $\exists v : U$, where U is a temporal logic formula, helped me to understand issues related to this dissertation.

My fellow graduate students Amitabh Shah, Pat Stephenson, and Micah Beck taught me many things. They always had time to discuss my work and to help me choose the right tools for programming and writing. Amitabh read an early draft of this work and his suggestions and comments were very helpful. For his hospitality during my last month in Ithaca, I am also deeply thankful. The help of David Loshin, who did not let me stop working on a problem that motivated this dissertation, is appreciated.

Ken Birman supported me during the initial stages of my research, providing me time to find my direction. I appreciate the confidence he has shown in me. I am also grateful to Efraim Arazi who let me have the time I needed to complete this research.

This work would not have been possible without the support of friends and family. My friends, Paz and Avi Kribus, always listened, gave advice, and showed a positive point of view. The devoted help of my in-laws, Yona and Dov Hefetz, was indispensable. My sister Rina Alcalay and her family relieved me from duties back home, allowing me freedom to pursue my goals.

I would like to thank my dear father, Aba Aizikowitz, for the support, advice, and encouragement that he has always given me. My children always reminded me that there is more to life than research. Finally, I deeply appreciate the help, encouragement, and advice from my wife Nava. Without her understanding and patience this work would have been impossible.

Support for this work was provided in part by the office of Naval Research under contract N00014-86-K0092, the National Science Foundation under Grant No. CCR-8701103, Digital Equipment Corporation, and Scitex America Corporation. Any opinions, conclusions or recommendations presented in this work are mine and do not reflect the views of these organizations.

Table of Contents

1 Introduction 1
- 1.1 Implementing Services . . . 3
- 1.2 Implementations from Refinement Mappings . . . 6
  - 1.2.1 Hiding Details . . . 9
- 1.3 Organization of Thesis . . . 10

2 Specifications and Implementations 11
- 2.1 Specifications . . . 11
  - 2.1.1 States . . . 12
  - 2.1.2 Actions . . . 13
  - 2.1.3 Behaviors . . . 14
- 2.2 A Simple Specification . . . 16
- 2.3 Programs are Specifications Too . . . 17
  - 2.3.1 Associating Control Variables with Statements . . . 19
- 2.4 External and Internal Variables . . . 19
  - 2.4.1 Externally-visible Parts of Behaviors . . . 21
- 2.5 Implementations . . . 22
  - 2.5.1 Implementation in terms of the Behavior Axioms . . . 25

3 Proof Outlines as a Specification Language 28
- 3.1 Program Proof Outlines . . . 29
  - 3.1.1 Actions of a Proof Outline . . . 30
  - 3.1.2 Behaviors of a Proof Outline . . . 31
- 3.2 Defining 'Implements' for Proof Outlines . . . 34
- 3.3 Structural Refinements . . . 36
- 3.4 Alternative Mapping for Control Variables . . . 47
- 3.5 A Remark on Structural Refinement . . . 47

4 Designing Distributed Implementations of Services 49
- 4.1 A Mutual Exclusion Service . . . 49
  - 4.1.1 A Single-Server Implementation . . . 50
  - 4.1.2 Deriving Multiple-Server Implementations . . . 51
- 4.2 A Methodology for Designing Distributed Services . . . 61

4.2.1 Performance 65
4.3 A Distributed Set 66
4.3.1 A Single-Server Implementation 66
4.3.2 A Multiple-Server Implementation 66

5 Conclusion **72**
5.1 Relation to Other Work 74
5.2 Future Work 77
5.2.1 Experience 78
5.2.2 Liveness 78
5.2.3 Failures 79

Bibliography **80**

List of Figures

1.1 A client/service program. 2
1.2 A single-server implementation. 3
1.3 A multiple-server implementation. 4
1.4 Client c_p with an in-line expansion of an implementation of the guard service. 7
1.5 A refinement of Figure 1.4 8

2.1 A specification H. 17
2.2 A specification in a generic assembly language. 18
2.3 Incrementing x by 2 25

4.1 A single-server implementation of mutual exclusion. 50
4.2 Translated assertions. 53
4.3 A first step in refining $enter_p$. 55
4.4 A second step in refining $enter_p$. 56
4.5 A third step in refining $enter_p$. 56
4.6 A **cobegin** refinement of $enter_p$. 57
4.7 Refining $exit_p$. 58
4.8 A distributed mutual exclusion. 59
4.9 A background update of server's state (gossip). 60
4.10 A multiple-server implementation of the mutual exclusion service. 60
4.11 A multiple-server implementation of *insert*. 68
4.12 A multiple-server implementation of *member* for $y = x \in S$. . . . 69
4.13 A multiple-server implementation of *member* for $y \Rightarrow x \in S$. . . . 70

Chapter 1

Introduction

Programs for distributed systems are often structured in terms of *clients* and *services*. A service supports some data abstraction, which consists of data objects and operations to manipulate them. Clients of the service are processes that use this abstraction. A *service-interface* defines the data abstraction, and a *service-implementation* is a realization of the abstraction. For example, the program in Figure 1.1 consists of two services, *tickets* and *guard*, and clients $c_1, \ldots, c_n$. The *tickets* service provides the abstraction of a ticket dispenser, supporting a single operation *getTicket*. The *guard* service provides the abstraction of an access guard, supporting the operations *enter* and *exit*. By invoking *getTicket*, a client acquires a unique ticket; by invoking *enter* with this ticket, exclusive access to the critical section is ensured.

Partitioning a service into a service-interface and a service-implementation allows clients to ignore implementation details of the service and allows the service-implementation to ignore implementation details of the clients. However, knowledge of how clients use a service often can be exploited in designing a service-implementation that is tailored to an application. For example, for the system of Figure 1.1, the fact that *exit* is invoked only by a client having access to the critical section simplifies the service-implementation—there is no need to

```
service tickets
    interface:
        type: ticket;
        operation:
            getTicket(var t : ticket);
    implementation ...
end of service tickets

service guard
    interface:
        operation:
            enter(t : ticket) returns boolean,
            exit();
    implementation ...
end of service guard

cobegin ||_{1≤p≤n}
c_p:  var tkt_p of ticket;
      do true →
          NCS_p
  get_p:  getTicket(tkt_p);
enter_p:  do ¬enter(tkt_p) → skip od;
          CS_p
 exit_p:  exit();
      od
coend
```

Figure 1.1: A client/service program.

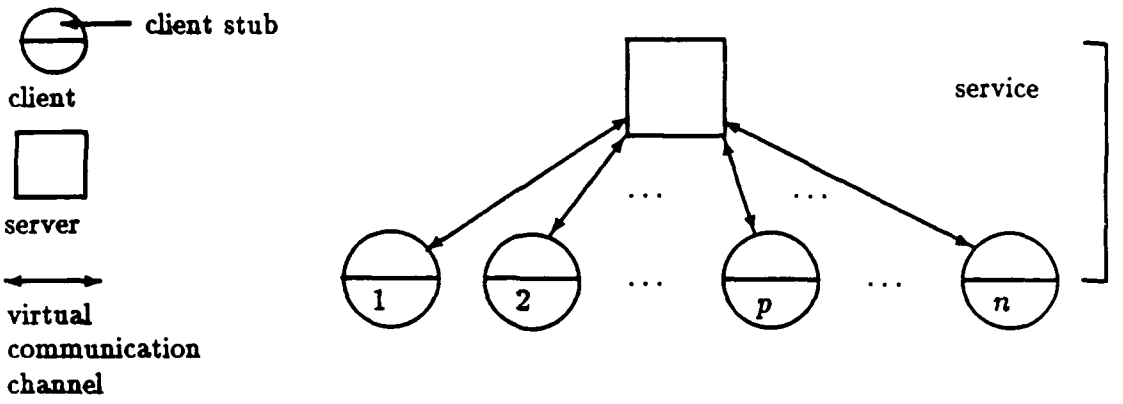

Figure 1.2: A single-server implementation.

test for spurious invocations, and concurrent invocations of *exit* never happen.

1.1 Implementing Services

A service is typically implemented by *servers*, which are processes that might reside anywhere in the distributed system, and by *client stubs*, which are programs that reside on computers executing clients. A server maintains state information pertaining to the implementation of the service. It receives *requests* to update or read that state and responds by sending *replies*. A client stub translates invocations of service operations into requests to one or more servers, sends these requests to the servers, receives the replies, and uses these replies to compute result value, which is returned to the client. For example, a server for the tickets service of Figure 1.1 might maintain a variable g to store the value of the most recently dispensed ticket and might support requests to increment g and to read its value. A client stub would issue such requests in implementing *getTicket*.

A service implementation that is based on a single server is structured as in Figure 1.2. Such an implementation is usually quite simple. However, it can have drawbacks. First, the service can only be as reliable as the computer that hosts

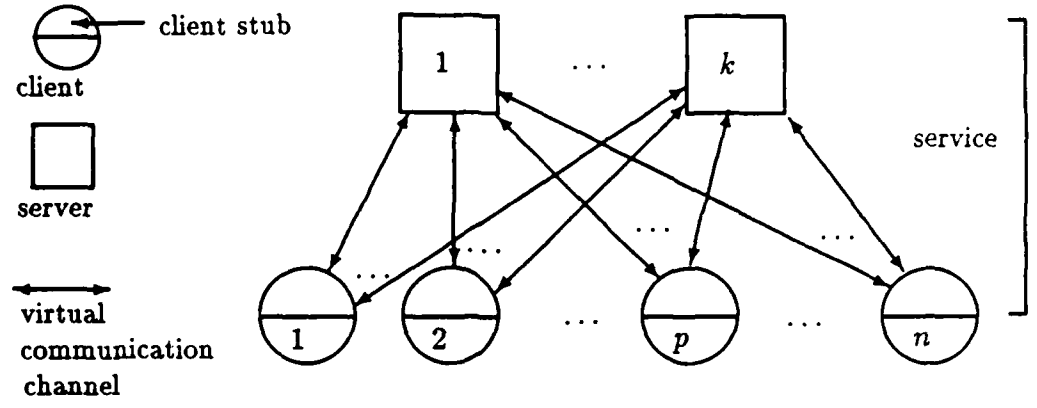

Figure 1.3: A multiple-server implementation.

the (single) server.[1] Failure of the server's computer halts the service. Second, having only a single server might lead to poor performance. For example, the computer executing the (single) server might not be able to handle requests in a timely manner, or, due to distance, some clients might only have a narrow bandwidth communication channel to the server's computer. Finally, organizational constraints might dictate that data maintained by the service reside at different sites, say for reasons of privacy or security. This would prevent any single site from hosting the server.

The problems associated with a single-server implementation can be circumvented by employing several servers to implement a service (see Figure 1.3). With multiple servers, failure of a computer that hosts a server can be masked. Performance problems can be avoided by using multiple servers and designing client stubs to forward requests to a server that is nearby; servers must update each other periodically [LL86]. Finally, organizational constraints can be addressed with multiple servers by assigning different servers to different parts of the organization.

Unfortunately, designing multiple-server implementations can be difficult. When several servers are used to implement a service, server activities might

[1] We assume clients do not fail.

have to be coordinated. For example, in a multiple-server implementation of the guard service of Figure 1.1, servers must be coordinated so that they do not concurrently grant different clients access to the critical section.

The state machine approach [Lam78,Sch86] is one of the more general methods for designing multiple-server implementations. Given a single-server implementation of a service, this approach permits derivation of a multiple-server implementation that, as far as clients can tell, behaves exactly like the single-server one. This is done by starting all the servers in the same initial state, using protocols to guarantee that every non-faulty server (i.e., server that runs on a non-faulty machine) behaves like the single server of the corresponding single-server implementation, and by requiring that client-stubs generate a result value using a reply from a non-faulty server.

The state machine approach exploits an assumption that the behavior of each server is completely determined by its initial state and the sequence of requests it processes. It is based on the following rules for processing requests:

Input agreement: All non-faulty servers receive every request.

Input order: All non-faulty servers process requests in the same order.

Output select: A client-stub generates a result value using a reply from a non-faulty server.

Input agreement, Input order, the requirement that all servers start with the same initial state, and the assumption that a server's behavior is deterministic, together guarantee that the states of all servers are identical after processing the ith request. Moreover, these server states will all be equal to the state of the single server of the corresponding single-server implementation. This is the reason that Output select permits a result value to be generated from the reply of any non-faulty server—any such reply is necessarily identical to the reply the corresponding single-server would have sent.

The state machine approach allows a multiple-server implementation to be derived mechanically from a single-server one. However, this multiple-server implementation may exhibit poor performance. This is because to guarantee that the sequences of requests processed by each server are identical, requests are tagged with unique identifiers from some total order (such as unique time-stamps), and servers process requests in that total order. If a request arrives at a server out of order, then the server must delay processing that request until it receives and processes all preceding requests. Thus, the state machine approach may cause requests that could be processed to be delayed.

A more fundamental problem with the state machine approach is that trade-offs between Input agreement, Input order, and Output select are not supported, although they are sometimes possible. Schemes where for each service operation, a stub requests that only a majority of the servers do the operation violate Input agreement, yet they are sometimes sufficient. For example, in an update operation, only a majority of servers need be changed provided that in read operations, replies from majority of servers are collected and one with the most recent information (according to some version identification scheme) is selected [Gif79, Tho79]. Thus, a weaker (and cheaper) Input agreement rule might be sufficient provided a stronger (and more expensive) Output select rule is used. Similarly, tradeoffs between Input order and Output select are sometime possible. Finally, if operations commute, then processing them in different orders will not generate different states; for such services Input order is too strong a requirement.

1.2 Implementations from Refinement Mappings

This thesis describes a methodology for designing multiple-server implementations given single-server implementations. The methodology is general enough to support trade-offs like the ones between Input agreement, Input order, and

```
c_p:   do true →
           NCS_p
  get_p:   getTicket(tkt_p);
  enter_p: do tkt_p ≠ nxt → skip od;
           CS_p
  exit_p:  nxt := nxt + 1;
       od
```

Figure 1.4: Client c_p with an in-line expansion of an implementation of the guard service.

Output select described above, but retains the simplicity of the state machine approach. In addition, the methodology supports design of service-implementations particularly well-suited for specific applications.

The key to our methodology is viewing a single-server implementation as a specification of a service, and regarding a multiple-server implementation to be correct if and only if it implements that specification. The methodology is based on defining a mapping that allows states of a multiple-server implementation to be regarded as states of a single-server one, and on showing that under this mapping the multiple-server implementation can be viewed as the single-server implementation. Having states of different servers be identical after processing the ith request, as required by the state machine approach, is only one mapping between states of a multiple-server implementation and states of a single-server one. Other mappings, when possible, lead to other implementations.

To illustrate the methodology, suppose a multiple-server implementation of the guard service of Figure 1.1 is desired, and we are given the single-server implementation described by $enter_p$ and $exit_p$ of Figure 1.4. In that implementation, a variable nxt indicates the value of the ticket held by the process granted access to the critical section. The server provides an operation *read* and an operation *increment* to access and modify nxt. Thus, in response to an invocation of *enter*, the stub requests *read* from the server, waits for the reply, compares the reply

```
c_p:   do true →
           NCS_p
  get_p:   getTicket(tkt_p);
  enter_p: do tkt_p ≠ nxt_p → skip od;
           CS_p
  exit_p:  ⟨cobegin ‖_{1≤j≤k}
                nxt_j := nxt_j + 1;
           coend⟩
       od
```

Figure 1.5: A refinement of Figure 1.4.

value with the ticket value supplied by the client, and returns the comparison result to the client. In response to *exit*, the client stub requests *increment*.

Given this single-server implementation, we can obtain a multiple-server implementation as follows. Suppose n servers are desired, where the jth server maintains variable nxt_j. Define a mapping that gives a value to nxt in terms of $nxt_1, \ldots, nxt_n$:

$$nxt = \begin{cases} nxt_1 & \text{if } true \\ \ldots & \ldots \\ nxt_n & \text{if } true \end{cases} \tag{1.1}$$

Observe that mapping (1.1) is well-defined only if

$$nxt_1 = nxt_2 = \ldots = nxt_n \tag{1.2}$$

holds on any state. So, (1.1) is a function only if (1.2) holds. Consequently, provided (1.2) holds, (1.1) allows us to regard the state of a multiple-server implementation as being a state of the single-server one.

It is straightforward to ensure that (1.2) holds initially. In order to ensure that (1.2) continues to hold, we must guarantee that whenever the state of one server is changed, then so are the states of all others. An implementation where this is done appears in Figure 1.5. Note that the only changes to Figure 1.4 are in $enter_p$ and $exit_p$, which now access variables $nxt_1, \ldots, nxt_n$ instead of

nxt. In $enter_p$ the stub requests that the local server (which maintains nxt_p) perform a read, and in $exit_p$ the stub atomically requests that all servers perform an increment—an action necessary to maintain (1.2). From the point of view of performance, mapping (1.1) provides an inexpensive read operation, but, to maintain (1.2) expensive updates are necessary.

Note that the multiple-server implementation presented in Figure 1.5, is, in fact, a state-machine style solution. However, we derived this implementation not by the rules of the state machine approach, but from a mapping. Another mapping, might have led to a different implementation.

1.2.1 Hiding Details

In the examples above, we have abstracted details concerning communication between stub and server. For example, the action $nxt := nxt + 1$ abstracts an interleaving of stub actions

$sendToServer$("increment");
$receiveFromServer$();

and server actions

$receiveFromStub(command)$;
if $command$ = "increment" $\rightarrow$ $nxt := nxt + 1$;
[] $command$ = "read" $\rightarrow$ $sendToStub(nxt)$;
fi

Also, the construct $\langle \ldots \rangle$ in action $exit_p$ of Figure 1.5 implies that all update requests that originate from a single *exit* invocation of client p, are performed in the same order by every server. To use another term from the literature, action $exit_p$ specifies an atomic broadcast of the requests to update the nxt_j's.

Using assignment statements to model interleaving of requests, replies, and server internal operations, and using a **cobegin** statement to model multicasting,

is just a consequence of the level of abstraction we chose for our programs. Our methodology is not limited to this level of abstraction; a designer should choose abstractions that are proper for the problem at hand. The difficulty of designing multiple-server implementations of a service, however, is mainly in designing a single-server implementation and in choosing a good mapping. Details of the communication are usually secondary. Choosing the right level of abstraction—one that reveals enough details so that the solution is not trivial, and hides just enough so that the solution is elegant—remains an art.

1.3 Organization of Thesis

Chapter 2 defines specifications and the notion of implementing a specification. Chapter 3 shows how proof outlines can be viewed as specifications and gives a theoretical foundation for a methodology of deriving multiple-server implementations from single-server implementations. In Chapter 4 we describe the methodology and illustrate it by examples. We conclude in Chapter 5 by relating our approach to the existing research. We also outline directions for applying and extending this work.

Chapter 2

Specifications and Implementations

In chapter 1, we suggested that a multiple-server implementation of a service can be viewed as an implementation of a specification given by some other—perhaps single-server—implementation. In this chapter, precise definitions of *specification* and *implementation* are given. The material is based on [Lam83,Lam89].

2.1 Specifications

A specification characterizes expected behavior of a system. To specify a system that comprises a single process, input/output relations, which relate values of variables at termination to their initial values, often suffice. For a system of concurrent processes, other properties, involving intermediate states of the computation, can be of interest. In fact, the program state at termination might be wholly irrelevant for a concurrent program, because many such programs are not intended to terminate. Thus, a specification of a concurrent system must define a set of sequences of states rather than simply a set of pairs of states.

Many different languages exist for expressing specifications. There are specially designed specification languages such as Larch [GHW85] and Z [Spi89],

terse mathematical notations, such as Temporal Logic [MP81], and graphical notations such as state-transition diagrams [HPSS87]. Programming languages can also serve as specification languages, since a program defines a set of sequences of states—those sequences that correspond to executions of the program. Independent of the language in use, the meaning of a specification of a concurrent system S can be described in terms of a triple $(\Sigma_S, \mathcal{A}_S, \mathcal{B}_S)$, where Σ_S is a set of *states*, $\mathcal{A}_S$ is a set of *actions*, and $\mathcal{B}_S$ is a set of sequences of states called *behaviors*.

2.1.1 States

States are functions that map *variable names* to *values*. We use $s(v)$ to denote the value of a variable v in a state s. The domain of every state in Σ_S is the set $N(S)$; the range of every state is the union of the types of the variables in $N(S)$. For any variable $v \in N(S)$, if v is of type T_v then $s(v) \in T_v$ holds for any state s because, a state always maps a variable name to a value in that variable's type. For convenience, we define $s(E)$ for an expression E, to be the value of E after every free variable v in E has been replaced by $s(v)$.

Control variables are used in order to specify aspects of the state that restrict possible transitions of control. We associate with every action a unique pair of distinct *control points*: an *entry* control point, and an *exit* control point. Before an action can be executed, its entry control point must be *active*; when it terminates, its exit control point becomes *active*. We assume that for every action $a \in \mathcal{A}_S$, $N(S)$ includes two Boolean control variables. The first, $at(a)$, is assigned the value *true* by any state for which the entry control-point of action a is active, and *false* by any other state; the second control variable, $after(a)$, is assigned *true* by any state for which the exit control-point is active, and *false* by any other state.

Not all functions with domain $N(S)$ represent meaningful assignments of values to variables. For example, it is reasonable to assume that the entry and exit

control points of an action a are never simultaneously active. Thus, a function that assigned *true* to both $at(a)$ and $after(a)$ would not be meaningful as a state. To exclude from Σ_S such meaningless functions, we specify *constraints*—Boolean expressions over $N(S)$. Only functions that satisfy these constraints are included in Σ_S. Thus, if x, y, z are in $N(S)$ and $x < y + z$ is a constraint, then, for every $s \in \Sigma_S$, the expression $s(x) < s(y) + s(z)$ must evaluate to *true*.

For convenience, we distinguish constraints of the form $x = E$, where x is a variable and E is an expression involving variables. We call such constraints *definitions* because they define the value that a state will assign to x in terms of the values assigned by that state to the variables in E. Variables that appear only on the right-hand side of definitions are considered *primitive*. Variables that appear on the left-hand side of a definition are considered *derived*, as their value is a function of the value assignment to the primitives. One case where derived control variables are handy is in arguing about execution of non-atomic actions. This is illustrated in Section 2.3.

2.1.2 Actions

An *action* is the basic unit of execution and changes the state indivisibly. Possible state changes of an action $a \in \mathcal{A}_S$ define a relation $\mathcal{R}_a^S \subseteq \Sigma_S \times \Sigma_S$ such that $(s, t) \in \mathcal{R}_a^S$ holds iff

- it is possible to execute a in state s, and
- executing a on s may result in state t.

An action $a \in \mathcal{A}_S$ can be specified by a predicate P_a^S that characterizes $\mathcal{R}_a^S$. The free variables in P_a^S are from $N(S)$ and a primed copy of $N(S)$, denoted $N'(S)$. A pair of states (s, s') satisfies P_a^S, denoted $(s, s') \models P_a^S$, iff the Boolean expression that results from replacing the free variables from $N(S)$ by their values in s and replacing the free variables from $N'(S)$ by their values in s', evaluates

to *true*. Thus, a primed variable in P_a^S refers to the value of that variable after the execution of a. The relation $\mathcal{R}_a^S$ is defined as

$$\begin{aligned}\mathcal{R}_a^S \;=\; \{(s,s') \in \Sigma_S \times \Sigma_S \mid ((s,s') \models P_a^S) \;\wedge \\ ((x \in N(S) \;\wedge\; x \notin \mathit{free}(P_a^S)) \Rightarrow s(x) = s'(x))\}.\end{aligned}$$

where $\mathit{free}(P)$ is the set of variables appearing free in the predicate P. If x and x' are free in P_a^S and there exists a pair $(s,s') \in \mathcal{R}_a^S$ for which $s(x) \neq s'(x')$ then we say that a *modifies* x. For example, an action $a \in \mathcal{A}_S$ that increments a variable x by 2 is specified by the predicate

$$P_a^S :\; \mathit{at}(a) \;\wedge\; \neg \mathit{after}(a) \;\wedge\; \neg \mathit{at}(a)' \;\wedge\; \mathit{after}(a)' \;\wedge\; x' = x + 2.$$

The action a, as specified, modifies $\mathit{at}(a)$, $\mathit{after}(a)$, and x.

An action a is *self-disabling* if for every $(s,t) \in \mathcal{R}_a^S$, there is no $u \in \Sigma_S$ such that $(t,u) \in \mathcal{R}_a^S$. Thus, after a completes execution, another action must be executed before a can execute again. Henceforth, we restrict consideration to self-disabling actions.

2.1.3 Behaviors

A *behavior* (of a specification S) is an infinite sequence $\sigma \in \Sigma_S^\infty$, where for every i, $\sigma[i]$ is a state,[1] and either $\sigma[i] = \sigma[i+1]$ or there exists an $a \in \mathcal{A}_S$ such that $(\sigma[i], \sigma[i+1]) \in \mathcal{R}_a^S$. A *property* is a set of behaviors. We assume that the property defined by a specification S, denoted $\mathcal{B}_S$, satisfies two restrictions. The first restriction is that $\mathcal{B}_S$ be *closed under stuttering*. The term *stuttering* refers to finite repetition of a state in a behavior. Let $\natural(\sigma)$ denote the state sequence that is obtained by replacing every subsequence of identical states by a single instance of the stuttered state. In other words, $\natural(\sigma)$ is stutter-free. Requiring $\mathcal{B}_S$ to be closed under stuttering means that if $\sigma \in \mathcal{B}_S$ holds, then for all σ' such that

[1] If $\sigma = s_0 s_1 \ldots$, then $\sigma[i] = s_i$, $\sigma[i..] = s_i s_{i+1} \ldots$, and $\sigma[i..j] = s_i s_{i+1} \ldots s_j$.

$\natural(\sigma) = \natural(\sigma')$ holds, $\sigma' \in \mathcal{B}_S$ holds as well. This requirement reflects the fact that an environment cannot distinguish between two successive identical states, and thus two behaviors that differ only by stuttering should be considered equivalent. In addition, as we shall see, having $\mathcal{B}_S$ closed under stuttering allows for a nice correspondence between a high-level action and low-level actions that implement it.

The second restriction on $\mathcal{B}_S$ is that it be *tail closed*. A set of sequences T is tail-closed if for every $\sigma \in T$, $\sigma[1..] \in T$ holds. Requiring $\mathcal{B}_S$ to be tail-closed guarantees that if σ is an element of $\mathcal{B}_S$ then so is $\sigma[i..]$ for every i. Tail-closure is consistent with the way machines execute programs: execution of a machine is based entirely on the machine's current state. Thus, if $\sigma[0..]$ is a behavior, then it is evidence that, for every i, if the specified program were started executing from state $\sigma[i]$, it would continue through all the states in $\sigma[i+1..]$, making $\sigma[i..]$ a behavior as well. Technically, tail-closure is helpful if one wants to consider behaviors as models for temporal logics that have a $\Box$-Generalization rule [MP81].

We describe $\mathcal{B}_S$ by conjunction Ax_S of *behavior axioms*. An infinite sequence of states σ is in $\mathcal{B}_S$ if and only if it satisfies all of these axioms. Formally,

$$\mathcal{B}_S = \{\sigma \in \Sigma_S^\infty \mid \sigma \models Ax_S\}.$$

Behavior axioms will be formulas of Temporal Logic, where the free variables are from $N(S)$.[2] Thus, $\mathcal{B}_S$ is the set of all *models* (from Σ_S^∞) of Ax_S.

The requirements that $\mathcal{B}_S$ be closed under stuttering and that it be tail closed impose restrictions on the behavior axioms. We consider only those Ax_S such that for every σ where $\sigma \models Ax_S$ holds, both $\natural\sigma \models Ax_S$ and $\sigma[1..] \models Ax_S$ hold as well. Also, we require that Ax_S be consistent with the specification of the individual actions. That is, if $\sigma \models Ax_S$, then, for every i, if $\sigma[i+1] \neq \sigma[i]$ then there exists an action $a \in \mathcal{A}_S$, such that $(\sigma[i+1], \sigma[i]) \models P_a^S$.

[2] We are assuming that the expressiveness of Temporal Logic [MP81] suffices for the behavior axioms.

In Section 2.1.1 we associated constraints with the set of states. However, being Boolean expressions over $N(S)$, constraints are Temporal Logic formulas and can be considered part of the behavior axioms. This leads to technically simpler specifications, and this is the approach we take.

2.2 A Simple Specification

As an example, we present a specification $(\Sigma_H, \mathcal{A}_H, \mathcal{B}_H)$ of a system H that increments an integer variable x by 2. The system consists of a single action α. Thus, $\mathcal{A}_H = \{\alpha\}$. From $\mathcal{A}_H$, we conclude that $at(\alpha)$ and $after(\alpha)$ must be elements of the domain of states. Thus, $N(H) = \{x, at(\alpha), after(\alpha)\}$, and Σ_H is the set of functions that map x to an integer value and $at(\alpha), after(\alpha)$ to Boolean values. Behavior axioms of H are:

- $at(\alpha) \Rightarrow at(\alpha) \mathsf{U} after(\alpha)$

  (If the entry control point of α is active, then it stays active unless the exit control point becomes active).

- $\neg(at(\alpha) \wedge after(\alpha))$

  (A constraint: the entry and exit control points of the action α are not active simultaneously).

- $at(\alpha) \vee after(\alpha)$

  (A constraint: either the entry control point of α or its exit control point are active).

- $after(\alpha) \Rightarrow \Box after(\alpha)$

  (If the exit control point of α is active then it stays active thereafter).

- $(\forall \mathrm{X} \in \mathcal{Z} : (at(\alpha) \wedge x = \mathrm{X}) \Rightarrow \Box(after(\alpha) \Rightarrow x = \mathrm{X} + 2))$

  (If the entry control point of α is active and x has an integer value X, then

state space: $N(H) = \{x, at(\alpha), after(\alpha)\}$

actions: $\mathcal{A}_H = \{\alpha\}$

behaviors: Specified by behavior axioms:

$$at(\alpha) \Rightarrow at(\alpha) \mathsf{U} after(\alpha)$$
$$\neg(at(\alpha) \wedge after(\alpha))$$
$$at(\alpha) \vee after(\alpha)$$
$$after(\alpha) \Rightarrow \Box after(\alpha)$$
$$(\forall \mathrm{X} \in \mathcal{Z} : (at(\alpha) \wedge x = \mathrm{X}) \Rightarrow \Box(after(\alpha) \Rightarrow x = \mathrm{X} + 2))$$
$$(\forall \mathrm{X} \in \mathcal{Z} : (after(\alpha) \wedge x = \mathrm{X}) \Rightarrow \Box x = \mathrm{X})$$

Figure 2.1: A specification H.

henceforth if the exit control point becomes active then the variable x will have the value X + 2).

- $(\forall \mathrm{X} \in \mathcal{Z} : (after(\alpha) \wedge x = \mathrm{X}) \Rightarrow \Box x = \mathrm{X})$
  (If the exit control point of h is active and the variable x has any integer value X, then x will have the value X thereafter).

Figure 2.1 summerizes this specification. Note that Ax_H satisfy the requirements that actions be self-disabling and that behaviors be both tail-closed and closed under stuttering.

2.3 Programs are Specifications Too

Programs can be regarded as defining specifications. As a specification, the meaning of a program P is $(\Sigma_P, \mathcal{A}_P, \mathcal{B}_P)$, defined as follows. The set of variables $N(P)$ consists of explicit as well as *implicit* variables of the program. Implicit variables are needed because when a programming language is used to specify actions, some aspects of the state become implicit. For example, the assignment statement α: $x := x + 2$ increments a variable x by 2, assigns *false* to the control variable $at(\alpha)$, and *true* to $after(\alpha)$. In α, only the variable x is explicit—both $at(\alpha)$ and $after(\alpha)$ are implicit variables of the action.

```
L:  lda x   /* load accumulator with x */
    ina     /* increment accumulator */
    ina
    sta x   /* store accumulator in x */
    hlt     /* halt */
x:  ds      /* define storage */
```

Figure 2.2: A specification in a generic assembly language.

The set of actions $\mathcal{A}_P$ consists of all atomic actions in P.

Finally, the behavior axioms Ax_P are encoded in the text of P and the semantics of the programming language.

As a simple example of how to regard a program as a specification, consider the following program H, given in a Pascal-like programming language.

H: **program**

var x: integer;

α: $x := x + 2$

end

As a specification, program H is identical to the specification H of Figure 2.1. Note, though, that $at(\alpha)$ and $after(\alpha)$ are explicit in Figure 2.1 and implicit in program H.

Program L of Figure 2.2, can also be regarded as specification. It is written in a generic assembler language and, naturally, specifies details that do not appear in a program of a high-level programming language. Program L has as its variables x, an accumulator, and control variables for each machine instruction. Every machine instruction defines an action. The axioms encode the effect of each. It should be obvious that programs L and H, and specification H, all specify similar behaviors. In Section 2.5, we formalize such relationships.

2.3.1 Associating Control Variables with Statements

When using programs to write specifications, one often needs to specify transfer of control between non-atomic as well as atomic statements. For that purpose, we associate with every statement T the control variables $at(T)$, $in(T)$, and $after(T)$. If T is not atomic, then the assignment of values to T's control variables should be consistent with the assignment of values to the control variables of the components of T. Thus, there are constraints on these control variables. Moreover, these constraints are of a special form—they define the value that a state assigns to control variables of T in terms of the values that the state assigns to the control variables of T's components. For this reason, the control variables of T are derived variables.[3] For example, a statement T: a ; b comprising of two atomic actions a and b, induces the following definitions for its derived control variables[4]:

$$\begin{aligned} at(T) &= at(a) \\ in(T) &= at(a) \vee at(b) \\ after(T) &= after(b) \end{aligned}$$

2.4 External and Internal Variables

In specifying a system, one must distinguish between *external* elements (i.e., names and actions), which are visible to the environment, and *internal* ones, which are invisible to the environment. For example, in the specification of the tickets and guard services of Figure 1.1, actions $getTicket()$, $enter()$, and $exit()$ are external; any name declared inside the implementation part of those services is internal. Ideally, one would like to specify a system in terms of what is observable by the environment—the externals. However, such specifications tend to be cumbersome, and thus internal elements are used as well. For example,

[3]For technical reasons, if T is the entire program then $after(T)$ is considered primitive.

[4]Assume T is not the entire program.

consider the specification of a set with operations to add and remove elements, and an operation to test for membership. It is possible to specify the effect of a test for membership operation in terms of the sequence of the preceding add and remove operations. However, it is much easier to maintain the contents of the set in some variable, having add and remove update this variable, and test search it. Such a variable though, is not (directly) visible to the environment, which can access it only by invoking the externally visible operations add, remove, and test. Thus, the variable is internal.

We use $(\overline{v}).S$ to denote a specification S in which $\overline{v}$ is a list of the internal names. The behaviors of a specification $(\overline{v}).S$ are described by $(\exists \overline{v} : Ax_S)$. The meaning of $\sigma \models (\exists x : U)$, where σ is an infinite sequence of states and U a Temporal Logic formula having the property of stutter equivalence (as all our behavior axioms do), is given by

$\sigma \models (\exists x : U)$ iff there exists sequences of states σ' and σ'' such that

- $\natural\sigma = \natural\sigma'$ (σ' is a stuttering equivalent of σ),
- σ'' is an x-variant of σ', i.e., differs from σ' by at most the value assignments to x, and
- $\sigma'' \models U$.

Classifying the elements of a specification S into externals and internals partitions the set $N(S)$ of names (of variables) into the disjoint subsets $N^E(S)$—the external names—and $N^I(S)$—the internal names. It also partitions the set $\mathcal{A}_S$ of actions into the disjoint subsets $\mathcal{A}_S^E$—the external actions—and $\mathcal{A}_S^I$—the internal actions. These partitions are constrained, though. Control variables associated with control points of external actions are external, and control variables associated with control points of internal actions are internal. Formally, we define:

(2.1) Consistent ext/int specification: The partitioning of $N(S)$ and $\mathcal{A}_S$ into

$N^E(S), N^I(S)$, and $\mathcal{A}_S^E, \mathcal{A}_S^I$ is *consistent* iff, for every $a \in \mathcal{A}_S$:

$$a \in \mathcal{A}_S^E \Leftrightarrow \{at(a), after(a)\} \subseteq N^E(S).$$

As a simple example, consider specification H given in Section 2.2. A classification into externals and internals might be, for the variables $N^E(H) = \{x\}$ and $N^I(H) = \{at(\alpha), after(\alpha)\}$, and for the actions $\mathcal{A}_H^E = \Phi$ and $\mathcal{A}_H^I = \{\alpha\}$. This would be stated as the specification $(at(\alpha), after(\alpha)).H$. Classifying x as external and α as internal implies that the environment can observe the changes in x, but it cannot observe the transfer of control resulting from the action that changes x. This classification means that the environment cannot distinguish behaviors of specification H from behaviors specified by program L of Section 2.3—both have the same effect on variable x. Had we classified α as an external action in specification H, behaviors specified by L could be distinguished from those specified by H.

2.4.1 Externally-visible Parts of Behaviors

The distinction between external and internal elements of a specification introduces a distinction between a state and an *externally-visible* state. An externally-visible state is a state with domain restricted to $N^E(S)$ instead of $N(S)$. Thus, an externally-visible state provides a value assignment for the external variables only. The externally-visible part of state $s \in \Sigma_S$ is denoted by $s|_{N^E(S)}$. An externally-visible part of a behavior σ is the sequence of externally-visible states that results from reducing every state in σ to its externally-visible part. We use $\sigma|_{N^E(S)}$ to denote the externally-visible part of the behavior σ. Note that an externally-visible part of a behavior is not necessarily a behavior. A behavior might have successive states s, t such that, for some internal action b, $(s,t) \in \mathcal{R}_b^S$ and, for some external name x, $s(x) \neq t(x)$. When reduced to their externally-visible parts, the externally-visible states will still differ (at least on x) but, since

$at(b)$ and $after(b)$ are not in the domain of the externally-visible states, their control part will not reflect the execution of any action.

2.5 Implementations

We can now turn to the question of what it means for one specification to implement another. Consider two specifications $(\overline{w}).L$ (for "low-level") and $(\overline{v}).H$ (for "high-level"). Specification $(\overline{w}).L$ *implements* $(\overline{v}).H$ iff the externally-visible part of every behavior of $(\overline{w}).L$ is an externally-visible part of some behavior of $(\overline{v}).H$. Formally, we require that for every $\sigma \in \mathcal{B}_L$ there exists a $\sigma' \in \mathcal{B}_H$ such that

$$\natural(\sigma|_{N^E(L)}) = \natural(\sigma'|_{N^E(H)}).$$

Thus, $(\overline{w}).L$ implements $(\overline{v}).H$ if the externally-visible parts of σ and σ' are equal but for stuttering. Note that because this definition of implements is in terms of a reduction to externally-visible parts, $N^E(H) = N^E(L)$ is a necessary condition for L to be considered an implementation of H.

One method for proving that $(\overline{w}).L$ implements $(\overline{v}).H$ is to define a function

$$F_{LH} : \Sigma_L \rightarrow \Sigma_H,$$

and show that for every behavior $\sigma \in \mathcal{B}_L$, the sequence

$$F_{LH}(\sigma[0])F_{LH}(\sigma[1])\ldots$$

is a **behavior** in $\mathcal{B}_H$. If for every $s \in \Sigma_L$, the value assignment to the external **variables** provided by $F_{LH}(s)$ is identical to the one provided by s, (i.e., for every $x \in N^E(H)$, $s(x) = F_{LH}(s)(x)$), then one can establish the existence of the behavior σ' in the definition of implements simply by defining σ' to be $F_{LH}(\sigma)$, where

$$F_{LH}(\sigma) \stackrel{\text{def}}{=} F_{LH}(\sigma[0])F_{LH}(\sigma[1])\ldots.$$

The function F_{LH} is called a *refinement mapping*.

Unfortunately, Abadi and Lamport have proved that for arbitrary specifications L and H, where L implements H, such a refinement mapping is not guaranteed to exist [AL88]. However, they also proved that if L and H satisfy certain restrictions, then it is possible to augment L with additional internal variables $\bar{x}$ (called *auxiliary* variables) such that the resulting specification $L^{\bar{x}}$ describes a property that is isomorphic to the one described by L, and a refinement mapping from $\Sigma_{L^{\bar{x}}}$ to Σ_H does exist. We call $L^{\bar{x}}$ an *auxiliary variables augmentation* of L. Thus, assuming that specifications satisfy the restrictions given in [AL88],[5] we have the following definition:

(2.2) Implements: A specification $(\overline{w}).L$ implements a specification $(\overline{v}).H$ if and only if

1. $N^E(H) = N^E(L)$.
2. There exist $L^{\bar{x}}$, an auxiliary variables augmentation of L, and refinement mapping $F_{LH} : \Sigma_{L^{\bar{x}}} \rightarrow \Sigma_H$ such that:
   (a) for every $s \in \Sigma_{L^{\bar{x}}}$ and for every $x \in N^E(H)$, $s(x) = F_{LH}(s)(x)$,
   (b) $(\forall a, (s,t) : a \in \mathcal{A}_H^E \wedge (s,t) \in \mathcal{R}_a^{L^{\bar{x}}} : (F_{LH}(s), F_{LH}(t)) \in \mathcal{R}_a^H)$,
   (c) for every $\sigma \in \mathcal{B}_{L^{\bar{x}}}$, $F_{LH}(\sigma) \in \mathcal{B}_H$.

Note that if our specification language has the property that for an external action a of a specification S, all the free variables in P_a^S—the predicate that specifies $\mathcal{R}_a^S$—are external, then requirement 2b of Implements (2.2) follows from requirements 1 and 2a.

Assuming that our formal language is rich enough, a natural way to define a refinement mapping F_{LH} is by defining an expression over $N(L)$ for every variable name in $N(H)$.[6] Let m_{LH} be a function that maps every variable name in $N(H)$ to an expression over the variable names in $N(L)$. Such a function can be used

[5] The type of specifications of concern in this thesis satisfy these restrictions.

[6] For simplicity, in the following discussion we ignore the fact that F_{LH} may have to be defined in terms of an auxiliary variables augmentation of L.

to augment every state $s \in \Sigma_L$, adding the additional variable names of $N(H)$ to the domain of s. Given m_{LH}, refinement mapping F_{LH} is any function from Σ_L to Σ_H such that for every $s \in \Sigma_L$ and for every $x \in N(H)$,

$$F_{LH}(s)(x) = s(m_{LH}(x)). \tag{2.3}$$

In order for m_{LH} to describe a refinement mapping, it must both be a function and an identity function for external variable names. That is, for every $s \in \Sigma_L$, $v \in N(H)$, and $x \in N^E(H)$,

$$s(m_{LH}(v)) \text{ is uniquely defined, and} \tag{2.4}$$

$$m_{LH}(x) = x. \tag{2.5}$$

A function, like m_{LH}, from $N(H)$ to expressions over $N(L)$ that defines a refinement mapping, is also called a refinement mapping.

We can construct m_{LH} by first defining an expression over $N(L)$ for every primitive variable in $N(H)$, such that (2.4) and (2.5) are satisfied. Then, we define expressions over $N(L)$ for every derived variable in $N(H)$. Recall, a variable $d \in N(H)$ is derived because it appears on the left-hand side of a definition, say $d = E_d$, where E_d is an expression over $N(H)$. For every derived variable $d \in N(H)$, we define $m_{LH}(d)$ to be $m_{LH}(E_d)$, where m_{LH} is extended to expressions in the usual way (e.g., $m_{LH}(a \wedge b) \stackrel{\text{def}}{=} m_{LH}(a) \wedge m_{LH}(b)$). Since every derived variable is ultimately defined in terms of the primitive ones, this completes the definition of the mapping m_{LH}.

To illustrate use of a refinement mapping and Implements (2.2), consider the specification given by program L of Figure 2.3 and program H of Section 2.3. Assuming that $N^E(L) = N^E(H) = \{x\}$, one would expect L to implement H. Since the variable y is internal, the externally-visible part of a behavior of L shows only changes in x, and both H and L make the same changes—they increment x by 2. We will not give a full proof that L implements H, but will

```
L:  program
        var x, y: integer;
l0: y := x;
l1: y := y + 2;
l2: x := y
    end
```

Figure 2.3: Incrementing x by 2 ...

present m_{LH}—a refinement mapping that can be used in such a proof.

$$\begin{aligned} m_{LH}(at(\alpha)) &\stackrel{\text{def}}{=} at(l_0) \vee at(l_1) \vee at(l_2) \\ m_{LH}(after(\alpha)) &\stackrel{\text{def}}{=} after(l_2) \\ m_{LH}(x) &\stackrel{\text{def}}{=} x \end{aligned}$$

Note that if the action α were external, then by Consistent ext/int specification (2.1), $at(\alpha)$ and $after(\alpha)$ would have been externals too, and this m_{LH} would not work because it violates 2a of Implements (2.2). In fact, if α were external, then the only possible implementation of H is H itself.

2.5.1 Implementation in terms of the Behavior Axioms

Defining candidate refinement mappings and checking them by testing each behavior is often infeasible. Since behaviors are characterized by behavior axioms, testing whether a candidate refinement mapping shows that a specification L implements a specification H, can be done using the behavior axioms of L and H rather than the actual behaviors in $\mathcal{B}_L$ and $\mathcal{B}_H$.

Using behavior axioms, condition (2c) of Implements (2.2) is equivalent to[7]

$$(\forall \sigma \in \Sigma_L^\infty : (\sigma \models Ax_L) \Rightarrow (F_{LH}(\sigma) \models Ax_H)). \tag{2.6}$$

Similarly, using predicates instead of relations to define the meaning of actions, condition (2b) of Implements (2.2) is equivalent to

$$(\forall a, (s,t) : a \in \mathcal{A}_H^E \wedge (s,t) \models P_a^L : (F_{LH}(s), F_{LH}(t)) \models P_a^H). \tag{2.7}$$

[7]For simplicity we omit the auxiliary variables augmentation notation.

Assuming that our formal language is rich enough so that F_{LH} can be defined syntactically in terms of a function like m_{LH} above, we can use (2.3) to argue that equation (2.6) is equivalent to

$$(\forall \sigma \in \Sigma_L^\infty : (\sigma \models Ax_L) \Rightarrow (\sigma \models m_{LH}(Ax_H))) \tag{2.8}$$

$$= \quad (\forall \sigma \in \Sigma_L^\infty : \sigma \models (Ax_L \Rightarrow m_{LH}(Ax_H))) \tag{2.9}$$

$$= \quad \Sigma_L^\infty \models (Ax_L \Rightarrow m_{LH}(Ax_H)). \tag{2.10}$$

Similarly, using (2.3) and (2.5), equation (2.7) is equivalent to

$$(\forall a, (s,t) : a \in \mathcal{A}_H^E, \wedge (s,t) \models P_a^L : \ (s,t) \models m_{LH}(P_a^H)). \tag{2.11}$$

Thus, using the above equations we can restate Implements (2.2), as follows:

(2.12) m-Implements: A specification $(\overline{w}).L$ implements a specification $(\overline{v}).H$ *if and only if*

1. $N^E(H) = N^E(L)$,
2. There exist L^z, an auxiliary variables augmentation of L, and a mapping m_{LH} from expressions over $N(H)$ to expressions over $N(L^z)$ such that:
   - (a) For every $s \in \Sigma_{L^z}$, $v \in N(H)$, and $x \in N^E(H)$,
     - i. $s(m_{LH}(v))$ is uniquely defined, and
     - ii. $m_{LH}(x) = x$.
   - (b) $(\forall a, (s,t) : a \in \mathcal{A}_H^E, \wedge (s,t) \models P_a^{L^z} : \ (s,t) \models m_{LH}(P_a^H))$,
   - (c) $\Sigma_{L^z}^\infty \models (Ax_{L^z} \Rightarrow m_{LH}(Ax_H))$.

We should point out that under the assumption that the formal language we use to express m_{LH} is rich enough, definition Implements (2.2) and definition m-Implements (2.12) are equivalent.

Using the definition of m_{LH}, condition 2c of m-Implements (2.12) is equivalent to

$$\Sigma_L^\infty \models (Ax_{L^x} \Rightarrow (Ax_H)^{u,v,w,\ldots}_{m_{LH}(u),m_{LH}(v),m_{LH}(w),\ldots}) \tag{2.13}$$

where $u, v, w, \ldots$ denote the variable names in $N(H)$. By condition 2a of m-Implements (2.12), Equation (2.13) is equivalent to

$$\Sigma_L^\infty \models (Ax_{L^x} \Rightarrow (Ax_H)^{\bar{v}}_{m_{LH}(\bar{v})}) \tag{2.14}$$

where $m_{LH}(\bar{v})$ denotes the list $m_{LH}(v_1), m_{LH}(v_2), \ldots$. Assuming that internal variables in specification $(\overline{w}).L$ are named differently than the internals in $(\overline{v}).H$, we have, by definition of m_{LH}, that for any $v_j \in \bar{v}$, v_j is not free in $m_{LH}(v_j)$. Thus, by predicate logic, we infer that (2.13) is equivalent to

$$\Sigma_L^\infty \models (Ax_{L^x} \Rightarrow (\exists \bar{v} : \bar{v} = m_{LH}(\bar{v}) : Ax_H)). \tag{2.15}$$

By predicate logic, from Equation (2.15) we infer

$$\Sigma_L^\infty \models (Ax_{L^x} \Rightarrow (\exists \bar{v} : Ax_H)). \tag{2.16}$$

In general, Equations (2.15) and (2.16) are not equivalent. Thus, in the general case, given Equation (2.16), one has no guarantee that a function such as m_{LH} can be found so that (2.15) can be established. However, due to the fact that specifications considered by this work satisfy the requirements for completeness of [AL88], and the assumption that our formal language is rich enough to express the refinement mapping, (2.15) and (2.16) can be considered equivalent.

Chapter 3

Proof Outlines as a Specification Language

Proof outlines [Sch] are useful in reasoning about *safety properties* [Lam77]—sets of behaviors in which certain finite prefixes (regarded as "bad" things) are prohibited. Proof outlines can also be used in deriving programs that satisfy a given safety property.

In this chapter, we explain how to interpret a proof outline as a specification. We then detail conditions that imply one proof outline implements another. In general, these conditions can be complex. However, when one proof outline is structurally related to the other (as defined later in this chapter), these conditions can be simplified. This structural relationship between proof outlines along with a theorem characterizing when a proof outline that is structurally related to another proof outline implements it are the basis of our methodology for deriving multiple-server implementations of a service from single-server implementations of that service. That methodology is presented in Chapter 4.

3.1 Program Proof Outlines

A *proof outline* is a program annotated with *assertions* before and after every statement. Assertions are predicates in which the free variables are the explicit and implicit variables of the program.[1] In a proof outline $PO(S)$ for a program S, the assertion associated with the control point $at(T)$, where T is a statement in S, appears just before the statement T and is denoted $pre_{PO(S)}(T)$; the assertion associated with the control point $after(T)$ appears immediately after T and is denoted $post_{PO(S)}(T)$. The term *triple* denotes a proof outline of the form $\{P\}T\{Q\}$ where P and Q are assertions and T is a statement. An example of a proof outline appears in Figure 4.1.

A proof outline $PO(S)$ is valid iff for every execution σ of S, if σ starts in a state where assertions associated with active control points hold, then in every state of σ, assertions associated with active control points hold. When a proof outline $PO(S)$ that is valid is viewed as a specification, every execution of S that starts in a state where the assertions associated with active control points hold is a behavior of $PO(S)$. However, in viewing $PO(S)$ as a specification, we would also like to argue that executions of programs other than S be regarded as behaviors of $PO(S)$. As we already know, this can be achieved by classifying the variables and actions of a specification into externals and internals. Thus, generally, a proof-outline specification is denoted $(\overline{v}).PO(S)$, where $\bar{v}$ denotes the internal variables.

The meaning of $(\overline{v}).PO(S)$, is given by $(\Sigma_{PO(S)}, \mathcal{A}_{PO(S)}, \mathcal{B}_{PO(S)})$. The state space $\Sigma_{PO(S)}$ is the state space of program S, as characterized in Section 2.3. The set of actions $\mathcal{A}_{PO(S)}$ is determined from the statements of S in a way described in 3.1.1 below. The set of behaviors $\mathcal{B}_{PO(S)}$ depends on the behaviors specified by program S and the assertions of $PO(S)$; it is defined in 3.1.2 below. The

[1] For simplicity, we ignore the *rigid* variables of a proof outline. It is straightforward to include them.

definitions we give are language independent, as was the definition of Σ_S in Section 2.3.

3.1.1 Actions of a Proof Outline

The set $\mathcal{A}_{PO(S)}$ of actions specified by proof outline $PO(S)$ is the same as $\mathcal{A}_S$—the set of all atomic actions of program S. $\mathcal{A}_S$ is determined by the statements of S and how they are composed.

The semantics of a programming language generally defines certain statement types to be atomic. It also defines certain statement construction rules that can be used in composing statements to obtain new ones. For example, the atomic statements of Guarded Commands [Dij76] as extended in [Sch] are *assignment* and *skip*, and the statement construction rules correspond to *composition*(;), *alternation*(**if** ... **fi**), *iteration*(**do** ... **od**), and *concurrent composition*(**cobegin** ... **coend**). For a given language, let $Atom(T)$ be a Boolean function that is *true* if statement T is atomic, and let $G_1, \ldots, G_n$ denote the statement construction rules defined by the language, where $T = G_i(T_1, \ldots, T_k)$ denotes the fact that statement T is constructed of statements $T_1, \ldots, T_k$ by rule G_i. For example, we expect that $Atom(x := z + 1)$ holds, $Atom(x := z + 1; d := 5)$ does not hold, and that

$$T\text{: } x := z + 1 \text{ ; } d := 5$$

is the composition of T_1: $x := z + 1$ and T_2: $d := 5$, so we have $T = G_;(T_1, T_2)$.

Some statements are *guarded* and may execute in a state only if that state satisfies a guard. We use $B \rightarrow T$ to denote such a guarded statement, where B is the guard. The guarded statement α: $x > 0 \rightarrow x := x - 1$ denotes a statement that can execute in a state s only if both $s(at(\alpha))$ and $s(x) > 0$ hold. Since the semantics of $true \rightarrow T$ is identical to the semantics of T, we consider every statement to be guarded.

Composing $B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k$, using statement construction rule G_i, pro-

duces a statement $T = G_i(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k)$. In general, this composition produces an *implicit action*, $g_T = g_{G_i(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k)}$, possibly null, that is part of T and is used to coordinate the components of T. For example, the implicit action might be one that evaluates the guards of the component statements, and based on the results, selects a particular component for execution. We will assume that this implicit action is atomic. Thus, for any such g_T, $Atom(g_T)$ is *true*.

In Guarded Commands, the implicit action of the composition construction rule (i.e., $G_;$) is empty, whereas the implicit action associated with iteration construction rule (i.e., $G_{\mathbf{do}}$) is not empty. For

$$T = G_{\mathbf{do}}(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k),$$

the implicit action g_T is required to select a component guarded statement, say $B_i \rightarrow T_i$, such that B_i holds, and transfer control to T_i (i.e., $at(T_i)$ becomes *true*); if such $B_i \rightarrow T_i$ cannot be found, control is transferred to the statement immediately following T (i.e., $after(T)$ becomes *true*).

A statement or an implicit action is called an *executable unit*. We assume that a program is a statement, thus an executable unit. For an executable unit T, let $Comps(T)$ denote the set of the components of T,

$$Comps(T) = \begin{cases} \Phi & \text{if } Atom(T) \\ \{g_T, T_1, \ldots, T_k\} & \text{if } T = G_i(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k) \end{cases} \tag{3.1}$$

and let $Comps^*(T)$ denote the reflexive transitive closure of $Comps(T)$. The set $\mathcal{A}_S$ of the atomic actions of program S is defined as

$$\mathcal{A}_S = \{a \in Comps^*(S) \mid Atom(a)\}.$$

3.1.2 Behaviors of a Proof Outline

The behavior axioms of a proof outline $PO(S)$ guarantee that if $PO(S)$ is valid then every execution of S that starts in a state where assertions associated with

active control points hold, is a behavior of $PO(S)$. The behavior axioms involve *control axioms* CA_S and an *invariance axiom* $Inv(PO(S))$. The control axioms characterize the behaviors with respect to the control variables. They guarantee that, with respect to control, behaviors can be considered executions of a program with the same structure as S. For example, in an execution of a program, it is never the case that both the entry and exit control points of the same action are active simultaneously. Also, it is always the case that if the entry control point of an action is active then it stays active until that action is executed. The control axioms guarantee that behaviors satisfy the above, and other, control requirements.

The invariance axiom, denoted $Inv(PO(S))$, further restricts behaviors based on the assertions in $PO(S)$. Its purpose is to guarantee that states in a behavior satisfy assertions corresponding to the active control points in a state.

Formally, $\mathcal{B}_{(\overline{v}).PO(S)}$, the set of behaviors specified by $(\overline{v}).PO(S)$, is given by

$$\mathcal{B}_{(\overline{v}).PO(S)} = \{\sigma \in \Sigma_S^\infty \mid \sigma \models (\exists \overline{v} : CA_S \wedge Inv(PO(S)))\}. \tag{3.2}$$

In the following subsections, we present the control and invariance axioms in detail.

Control Axioms

The control axioms for a proof outline $PO(S)$ depend on the program S and the semantics of the programming language. Recall that in Section 2.1.3 we stated that constraints would be included in the behavior axioms. In keeping with that decision, the first two types of control axioms are those that constrain the assignment of values to control-variables.

Recall that definitions—formulas of the form $x = E$, where x is a variable and E is an expression involving variables—are a special type of constraint. For an executable unit T, let DEF(T) denote a set of definitions and let CNSR(T) denote

the set of remaining constraints. The set $\text{DEF}(T)$ is defined as

$$\text{DEF}(T) = \begin{cases} \{\text{“}in(T) = at(T)\text{”}\} & \text{if } Atom(T) \\ \{D_{at(T)}, D_{in(T)}\} & \text{if } T \text{ is the whole program} \\ \left\{ \begin{array}{l} D_{at(T)}, \\ D_{in(T)}, \\ D_{after(T)} \end{array} \right\} & \text{if } T = G_i(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k) \end{cases} \tag{3.3}$$

where D_x denotes the definition “$x = B_x$” and B_x is a Boolean expression over control variables. For any primitive control variable x we define D_x to be the null formula. Note that for a program T, according to (3.3), the control variable $after(T)$ is primitive. Let $\text{DEF}^*(T)$ denote the transitive closure of $\text{DEF}(T)$:

$$\text{DEF}^*(T) = \text{DEF}(T) \cup \Big(\bigcup_{Q \in Comps(T)} \text{DEF}^*(Q) \Big). \tag{3.4}$$

The control-variable definitions specified by a program S are the elements of $\text{DEF}^*(S)$.

The set $\text{CNSR}(T)$ of the remaining constraints of T, is defined as

$$\text{CNSR}(T) = \begin{cases} \Phi & \text{if } Atom(T) \\ \{B_l \mid l = 1 \ldots n_T\} & \text{otherwise} \end{cases} \tag{3.5}$$

where $B_1, \ldots, B_{n_T}$ are Boolean expressions over the control variables of $Comps(T)$ and, possibly, T. Let $\text{CNSR}^*(T)$ be defined by

$$\text{CNSR}^*(T) = \text{CNSR}(T) \cup \Big(\bigcup_{Q \in Comps(T)} \text{CNSR}^*(Q) \Big). \tag{3.6}$$

The set $\text{CNSR}^*(S)$ consists of the remaining constraints of program S.

The third type of control axioms are *persistence axioms*. A persistence axiom states that an entry control point that becomes active will stay active until its associated action executes. Only execution of an action a can deactivate control point $at(a)$ and activate control point $after(a)$. Formally, the persistence axiom for action a is $\Box(in(a) \Rightarrow (in(a) \mathsf{U} after(a)))$.

The fourth and final type of control axiom is the *termination axiom*. The termination axiom captures the fact that once the control point at the end of the program becomes active, it stays active thereafter. It is $\Box(after(S) \Rightarrow \Box(after(S)))$. A termination axiom allows finite executions (execution of terminating programs) to be viewed as infinite ones, where the state at termination is stuttered ad infinitum.

In summary, the control axioms of proof outline specification $PO(S)$ are:

CA 1 (definitions) *The set* $\text{DEF}^*(S)$

CA 2 (constraints) *The set* $\text{CNSR}^*(S)$

CA 3 (persistence) $\bigwedge_{a \in \mathcal{A}_S} \Box(in(a) \Rightarrow (in(a) \mathsf{U} after(a)))$

CA 4 (termination) $\Box(after(S) \Rightarrow \Box(after(S)))$

□

Invariance Axiom

An invariance axiom asserts the invariance of the *proof-outline invariant*

$$I_{PO(S)} : \bigwedge_{a \in \mathcal{A}_S} at(a) \Rightarrow pre_{PO(S)}(a) \ \wedge \ after(S) \Rightarrow post_{PO(S)}(S). \qquad (3.7)$$

Formula $I_{PO(S)}$ is the formal statement of the requirement that whenever a control point is active in a state, its corresponding assertion holds in that state. The **invariance axiom** restricts behaviors to be only sequences where every state satisfies **(3.7)**. Formally,

$$Inv(PO(S)): \Box I_{PO(S)}. \qquad (3.8)$$

3.2 Defining 'Implements' for Proof Outlines

We now turn to the question of what it means for one proof outline to implement another. Since proof outlines are specifications, the answer to this question is

given by Implements (2.2). However, in order to use logic for arguing 'implements' we use the equivalent definition, m-Implements (2.12). In the following, we discuss the details of using this definition when proof outlines are involved.

Consider two proof outline specifications: $(\overline{v}).PO(H)$, and $PO(L)$. According to m-Implements (2.12), in order to prove that $PO(L)$ implements $(\overline{v}).PO(H)$ one must show that conditions 1 and 2 hold.

Conditions 1 and 2a are straightforward.

In the case of proof outlines, we can show that 2b is redundant—it follows from 1 and 2(a)ii. To see this, observe that in proof outlines, actions are specified by programming language statements. If such a statement specifies an external action, then it would have to appear unchanged in any implementation. Thus, any internal variable mentioned in that statement would have to become part of every implementation. Internal variables that are so constrained are indistinguishable from external variables. So, when proof outlines are used as specifications, external actions must be specified by statements involving only external variables. We, therefore, conclude that for an external action $a \in \mathcal{A}_H^E$, all free variables of P_a^H are externals. This conclusion, together with $N^E(L) = N^E(H)$ (condition 1) implies that $P_a^L = P_a^H = P_a$. From condition 2(a)ii we infer that $P_a = m_{LH}(P_a)$, and thus conclude that condition 2b holds whenever 1 and 2(a)ii do.

Finally, in order to satisfy condition 2c, we must prove

$$(CA_L \wedge \Box I_{PO(L)}) \Rightarrow m_{LH}(CA_H \wedge \Box I_{PO(H)}). \tag{3.9}$$

However, showing that (3.9) holds might be difficult. In particular, since a variable name $x \in N^I(H)$ is mapped by m_{LH} to an expression over $N(L)$, any separation between control and invariance in behavior axioms $CA_H \wedge \Box I_{PO(H)}$ might disappear in $m_{LH}(CA_H \wedge \Box I_{PO(H)})$. Fortunately, if $PO(L)$ is not an arbitrary proof outline, but one where the structure of program L resembles the structure of program H, then m_{LH} can be constructed such that internal control variables

of H are defined only in terms of control variables of L. This simplifies the proof of (3.9) and provides a standard way to define m_{LH} for control variables, as we now see.

3.3 Structural Refinements

A program R is a *structural refinement* of a program A if the control structure of A is an abstraction of the control structure of R. We formalize this definition as:

(3.10) structural refinement of executable units: An executable unit T' is a *structural refinement* of an executable unit T iff

- $Atom(T)$, or
- $T = G_i(B_1 \rightarrow T_1, \ldots, B_k \rightarrow T_k)$, $T' = G_i(B'_1 \rightarrow T'_1, \ldots, B'_k \rightarrow T'_k)$, and for every $j \in 1 \ldots k$, T'_j is a structural refinement of T_j, or
- T' is an auxiliary elements augmentation of some T'', and T'' is a structural refinement of T.

For example, program L in Section 2.3 is a structural refinement of the program H in Section 2.3. This is due to the fact that H consists of a single atomic action α, thus any program will be a structural refinement of H. As another example, consider the following statement T,

T: **if** $x > 0 \rightarrow x := -x \;[]\; x \leq 0 \rightarrow$ **skip fi**

A structural refinement of T must be an if with two guarded statements. The guarded statements, however, can be somewhat more complex. For example, the following statement T' is a structural refinement of T:

T': **if** $x \geq 0 \rightarrow$ **do** $x > 0 \rightarrow x := -x$ **od**
[] $x \leq 0 \rightarrow x := x$
fi

Statement T'' below is not a structural refinement of T:

T'': **if** $x \geq 0 \rightarrow$ **do** $x > 0 \rightarrow x := -x$ **od**
$[]$ $x \leq 0 \rightarrow x := x$
$[]$ $x = 0 \rightarrow$ **skip**
fi

The following technical lemma relates $Comps^*(T')$ and $Comps^*(T)$, when T' is a structural refinement of T. It will be useful in stating our main results.

Lemma 1 *If T' is a structural refinement of T then there exists a total mapping $h : Comps^*(T) \rightarrow Comps^*(T')$ such that for any $c \in Comps^*(T)$, $h(c)$ is the element of $Comps^*(T')$ that is the structural refinement of c.*

Proof: By induction, using the definition of structural refinement. □

The definition of structural refinement for executable units can be extended to proof outlines as follows:

(3.11) Structural refinement of proof outlines: $(\overline{w}).PO(T')$ is a structural refinement of $(\overline{v}).PO(T)$ iff T' is a structural refinement of T, and, for every $a \in \mathcal{A}_T^E$, $h(a)$ is identical to a.

The significance of identifying structural refinements when proving that one proof outline implements another is captured by the following theorem. It shows that a predefined, generic, mapping of the control variables of $N(A)$ can be used in constructing a refinement mapping to prove that $(\overline{w}).PO(R)$, a structural refinement of $(\overline{v}).PO(A)$, implements $(\overline{v}).PO(A)$.

Theorem 2 (implements) *If $(\overline{w}).PO(R)$ is a structural refinement of $(\overline{v}).PO(A)$ and $N^E(R) = N^E(A)$, and if there exists a function m from $N(A)$ to $N(R)$ that satisfies the following conditions*

H-1 For every $x \in N^E(A)$:

$$m(x) = x$$

H-2 For every $a \in \mathcal{A}_A$:

$$m(at(a)) = in(h(a))$$
$$m(after(a)) = after(h(a))$$

H-3

$$m(after(A)) = after(h(A))$$

H-4 For every derived control variable $x \in N(A)$:

$$m(x) = m(B_x), \text{ where } "x = B_x" \in \text{DEF}^*(A)$$

H-5 If $e_1 \star e_2$ is an expression over $N(A)$ then

$$m(e_1 \star e_2) = m(e_1) \star m(e_2)$$

H-6

$$\bigwedge_{a \in \mathcal{A}_A}((I_{PO(R)} \wedge m(at(a))) \Rightarrow m(pre_{PO(A)}(a)))$$
$$I_{PO(R)} \wedge m(after(A)) \Rightarrow m(post_{PO(A)}(A))$$

then $PO(R)$ *implements* $(\overline{v}).PO(A)$.

In the proof of Theorem 2 (implements), we reason about control variables. To do so, we must assume that the semantic description of the programming language provides the necessary apparatus. Among other things, we must assume that the control axioms are strong enough so that we can prove certain true things about control aspects of the state. Since Theorem 2 (implements) is formulated in a manner that is independent of the particular programming language being used, we must make some assumptions about the logical apparatus available to us for reasoning about control variables.

The first assumption we make is that the set $\text{DEF}^*(S)$—the definitions of derived control variables of program S—allows every derived control variable to be expressed in terms of primitive control variables only. We say that the set $\text{DEF}^*(S)$ is *sufficient* iff for every derived control variable x, the expression B_x (in the definition D_x) can be reduced to an equivalent expression over only primitives, by repeatedly replacing derived control variables by their definitions in $\text{DEF}^*(S)$. Henceforth, we assume that $\text{DEF}^*(S)$ is sufficient.

The second assumption we make is that the control axioms imply that derived control variables are related properly to the control variables of the components of their statement.

(3.12) consistent control axioms: For every $T \in Comps^*(S)$, CA_S implies:

$$at(T) \Rightarrow in(T) \tag{3.13}$$

$$\bigwedge_{a \in Comps^*(T)} (in(a) \Rightarrow in(T)) \tag{3.14}$$

$$in(T) \Rightarrow \bigvee_{a \in Comps^*(T)} in(a) \tag{3.15}$$

$$\neg(in(T) \wedge after(T)) \tag{3.16}$$

$$in(T) \Rightarrow (in(T) \mathsf{U} after(T)) \tag{3.17}$$

The third assumption is that constraints (and definitions) reflect the structure of the program. Some aspects of control reflect the manner in which executable units have been composed. Thus, control axioms should be referentially transparent with respect to this composition. Formally,

(3.18) structure: For any statements T and T', such that $T = G_i(O_1, \ldots, O_k)$ and $T' = G_i(O'_1, \ldots, O'_k)$ (where O_i denotes $C_i \rightarrow T_i$), if $B \in \text{CNSR}(T)$ then $B^{T,O_1,\ldots,O_k}_{T',O'_1,\ldots,O'_k} \in \text{CNSR}(T')$. Similarly for $B \in \text{DEF}(T)$.

Our fourth and last assumption about the control semantics of the programming language restricts constraint-expressions—elements of $\text{CNSR}^*(S)$ or right hand sides of definitions in $\text{DEF}^*(S)$—for S. To understand the need for this assumption, recall that when $PO(L)$ is a structural refinement of $PO(H)$, an internal (atomic) action $a \in \mathcal{A}^I_H$ is structurally refined by $h(a) \in Comps^*(L)$, where $h(a)$ is not necessarily atomic. By structure (3.18), every constraint-expression of $PO(H)$ that involves a has a matching constraint-expression of $PO(L)$ where references to a are replaced by references to $h(a)$. This means that for any behavior of $PO(L)$, whenever control is either at the start of $h(a)$ or at the end

of $h(a)$, states are constrained to satisfy these matching constraint-expressions. When such states are viewed as states of $PO(H)$, they satisfy the constraint-expressions involving a. This, however, is not enough because we have decided that states in $PO(L)$ for which control is inside $h(a)$ will be viewed as states in $PO(H)$ for which control is at the start of a. Thus, a state of $PO(L)$ in which a control point inside $h(a)$ is active, must satisfy constraint-expressions involving $at(a)$ when viewed as a state of $PO(H)$. This leads to a requirement that if states of $PO(L)$ satisfy constraint-expressions involving $at(h(a))$ and $after(h(a))$ they must also satisfy the same constraint-expressions where every occurrence of $at(h(a))$ is replaced by $in(h(a))$. We call this property **coverage** since it implies that constraint-expressions hold for all the control points covered by $in(h(a))$. Note, for atomic $h(a)$ this requirement is trivial since $at(h(a)) = in(h(a))$.

(3.19) coverage: For any program S, if "$x = B$" is in $\text{DEF}^*(S)$ or B is a constraint in $\text{CNSR}^*(S)$, and if B' is any expression derived from B by replacing derived variables in B by the right hand side of their definition from $\text{DEF}^*(S)$, then[2]

$$(\text{DEF}^*(S) \wedge \text{CNSR}^*(S)) \Rightarrow (B' \Rightarrow (B')^{\overline{at}}_{\overline{in}}).$$

In proving Theorem 2 (implements) we use several lemmas, which we present here together with their proofs.

Lemma 3 (invariance) *If H-1 through H-6 hold, then the following formula is valid:*

$$I_{PO(R)} \Rightarrow m(I_{PO(A)})$$

Proof:

1 $I_{PO(R)} \Rightarrow m(I_{PO(A)})$

[2]The notation $E^{\overline{at}}_{\overline{in}}$ is a shorthand for $E^{at(u),at(v),\ldots}_{in(u),in(v),\ldots}$, where $at(u), at(v), \ldots$ denote all the at control variables in E.

1.1 $I_{PO(R)}$ assumption

1.2 Consider any $a \in \mathcal{A}_A$

1.3 $m(at(a)) \Rightarrow m(pre_{PO(A)}(a))$

1.3.1 $m(at(a))$ assumption

1.3.2 $I_{PO(R)} \wedge m(at(a))$ $\wedge$ - incl on 1.1, 1.3.1

1.3.3 $m(pre_{PO(A)}(a))$ MP H-6, 1.3.2

1.4 $\bigwedge_{a \in \mathcal{A}_A} m(at(a)) \Rightarrow m(pre_{PO(A)}(a))$ 1.2, 1.3

1.5 $m(\bigwedge_{a \in \mathcal{A}_A}(at(a) \Rightarrow pre_{PO(A)}(a)))$ 1.4, H-5

1.6 $m(after(A)) \Rightarrow m(post_{PO(A)}(A))$

1.6.1 $m(after(A))$ assumption

1.6.2 $m(post_{PO(A)}(A))$

1.6.2.1 $I_{PO(R)} \wedge m(after(A))$ $\wedge$-incl on 1.1, 1.6.1

1.6.2.2 $m(post_{PO(A)}(A))$ MP 1.6.2.1, H-6

1.6.3 $m(after(A)) \Rightarrow m(post_{PO(A)}(A))$ 1.6.1, 1.6.2

1.7 $m(\bigwedge_{a \in \mathcal{A}_A}(at(a) \Rightarrow pre_{PO(A)}(a))) \wedge (m(after(A)) \Rightarrow m(post_{PO(A)}(A)))$
$\wedge$-incl 1.5, 1.6

1.8 $m(\bigwedge_{a \in \mathcal{A}_A}(at(a) \Rightarrow pre_{PO(A)}(a)) \wedge (m(after(A)) \Rightarrow m(post_{PO(A)}(A))))$
H-5 on 1.7

1.9 $m(I_{PO(A)})$ 1.8, Equation (3.7)

□

Lemma 4 (unless) *If H-1 through H-6 hold, then the following formula is valid:*

$$\bigwedge_{b \in \mathcal{A}_A} \Box(in(h(b)) \Rightarrow (in(h(b)) \mathsf{U} after(h(b)))) \Rightarrow$$
$$\Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} m(after(b)))).$$

Proof:

1 Consider any $b \in \mathcal{A}_A$

2 $\Box(in(h(b)) \Rightarrow (in(h(b)) \mathsf{U} after(h(b)))) \Rightarrow$
$\Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} m(after(b))))$

2.1	$\Box(in(h(b)) \Rightarrow (in(h(b)) \mathsf{U} after(h(b))))$	assumption
2.2	$m(at(b)) = in(h(b))$	1 and H-2
2.3	$\Box(m(at(b)) = in(h(b)))$	$\Box$-generalization, 2.2
2.4	$\Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} after(h(b))))$	TL 2.1, 2.3
2.5	$m(after(b)) = after(h(b))$	1 and H-2
2.6	$\Box(m(after(b)) = after(h(b)))$	$\Box$-generalization, 2.5
2.7	$\Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} m(after(b))))$	TL 2.6, 2.4

3 By 2 and 1 conclude

$$\bigwedge_{b \in \mathcal{A}_A} \Box(in(h(b)) \Rightarrow (in(h(b)) \mathsf{U} after(h(b)))) \Rightarrow \Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} m(after(b)))).$$

□

The next lemma shows that m, as defined in Theorem 2 (implements), preserves the constraints of specification $PO(A)$. Thus, a constraint $B \in \text{CNSR}^*(A)$ is mapped by m to a formula $m(B)$ that holds at any state of a behavior of $PO(R)$.

Lemma 5 (constraints) *If the hypotheses of Theorem 2 (implements) hold, then for any constraint* $B \in \text{CNSR}^*(A)$, *the formula* $(\text{DEF}^*(R) \wedge \text{CNSR}^*(R)) \Rightarrow m(B)$ *is valid.*

Proof: For a definition "$x(T) = B_{x(T)}$" in $\text{DEF}^*(A)$, define a $D_{x(T)}$ *expansion step* to be substitution of the expression $B_{x(T)}$ for all the occurrences of a derived control variable $x(T)$, where x can be "*at*", "*in*", or "*after*", and T is an executable unit. Consider any constraint B in $\text{CNSR}^*(A)$. Let $D_{x(T_1)} \ldots D_{x(T_n)}$ be a sequence of expansion steps such that the ith step is applied to B^{i-1}, resulting in B^i, where $B = B^0$, and B^n mentions primitive control variables only. Since $\text{DEF}^*(A)$ is sufficient, such an expansion sequence exists.

From H-5 we infer

$$m(B) = \ldots = m(B^n). \tag{3.20}$$

From the fact that B^n is primitive, and from hypotheses H-2 and H-3 we get:[3]

$$m(B^n) = (B^n)^{at(\cdot),after(\cdot)}_{in(h(\cdot)),after(h(\cdot))}.$$

Observe that

$$(B^n)^{at(\cdot),after(\cdot)}_{in(h(\cdot)),after(h(\cdot))} = ((B^n)^{(\cdot)}_{h(\cdot)})^{at}_{in}$$

and thus, using (3.20), we have

$$m(B) = m(B^n) = ((B^n)^{(\cdot)}_{h(\cdot)})^{at}_{in}. \tag{3.21}$$

Given (3.21), our goal is to show

$$(\text{DEF}^*(R) \wedge \text{CNSR}^*(R)) \Rightarrow ((B^n)^{(\cdot)}_{h(\cdot)})^{at}_{in}. \tag{3.22}$$

To show (3.22) we first show that $B^{(\cdot)}_{h(\cdot)}$ is in $\text{CNSR}^*(R)$. Then, using

$$B^{(\cdot)}_{h(\cdot)} \in \text{CNSR}^*(R)$$

we will argue that

$$(\text{DEF}^*(R) \wedge \text{CNSR}^*(R)) \Rightarrow (B^n)^{(\cdot)}_{h(\cdot)}. \tag{3.23}$$

[3]We use the notation $(\cdot)$ for an action or statement name in any control variable. For example,

$$(at(a) \wedge at(b) \wedge after(T))^{at(\cdot),after(\cdot)}_{in(h(\cdot)),after(h(\cdot))} = at(h(a)) \wedge at(h(b)) \wedge after(h(T))$$

Finally, by coverage (3.19) we will conclude that (3.22)—our goal—holds. The details follow.

Due to (3.6), having B a constraint of A (i.e., $B \in \text{CNSR}^*(A)$) implies that, for some $T \in \mathit{Comps}^*(A)$, B is a constraint of T (i.e., $B \in \text{CNSR}(T)$). From the hypotheses of Theorem 2 (implements), R is a structural refinement of A. Therefore, by Lemma 1, we get $h(T) \in \mathit{Comps}^*(R)$ and that $h(T)$ is a structural refinement of T. From $B \in \text{CNSR}(T)$ and $h(T)$ a structural refinement of T, we infer, by structure (3.18), that $B^{(\cdot)}_{h(\cdot)} \in \text{CNSR}(h(T))$. Thus, from

$$h(T) \in \mathit{Comps}^*(R)$$

and (3.6) we infer

$$B^{(\cdot)}_{h(\cdot)} \in \text{CNSR}^*(R). \tag{3.24}$$

Now we show that from (3.24) and $\text{DEF}^*(R)$ it is possible to infer $(B^n)^{(\cdot)}_{h(\cdot)}$, thus proving (3.23). Recall that $D_{x(T_j)}$ denotes a definition in $\text{DEF}^*(A)$. By structural refinement, $h(T_j) \in \mathit{Comps}^*(R)$, and from structure (3.18) we can infer that $D_{x(h(T_j))}$ is a definition in $\text{DEF}^*(R)$. So, $D_{x(h(T_1))} \ldots D_{x(h(T_n))}$ are all definitions in $\text{DEF}^*(R)$, and they can be used to derive $(B^{(\cdot)}_{h(\cdot)})^n$ from $B^{(\cdot)}_{h(\cdot)}$. This proves

$$(\text{DEF}^*(R) \wedge \text{CNSR}^*(R)) \Rightarrow (B^{(\cdot)}_{h(\cdot)})^n. \tag{3.25}$$

By induction on n one can prove

$$(B^n)^{(\cdot)}_{h(\cdot)} = (B^{(\cdot)}_{h(\cdot)})^n. \tag{3.26}$$

From (3.25) and (3.26) conclude that (3.23) holds.

Finally, from coverage (3.19) and (3.23) we conclude that (3.22) is valid. □

Lemma 6 (termination) *If the hypotheses of Theorem 2 (implements) hold, then the formula*

$$(\mathit{after}(R) \Rightarrow \Box(\mathit{after}(R))) \Rightarrow (m(\mathit{after}(A)) \Rightarrow \Box(m(\mathit{after}(A))))$$

is valid.

Proof:

1 $(after(R) \Rightarrow \Box(after(R))) \Rightarrow (m(after(A)) \Rightarrow \Box(m(after(A))))$

1.1 $after(R) \Rightarrow \Box(after(R))$	assumption
1.2 $h(A) = R$	R is a structural refinement of A
1.3 $m(after(A)) = after(h(A))$	Hypothesis H-3
1.4 $m(after(A)) = after(R)$	1.2, 1.3
1.5 $m(after(A)) \Rightarrow \Box(m(after(A)))$	TL 1.1, 1.4

□

The proof of Theorem 2 (implements) has two major steps. In the first one, the above lemmas are used to show

$$(CA_R \wedge \Box I_{PO(R)}) \Rightarrow m(CA_A \wedge \Box I_{PO(A)}).$$

In the second step, the result of the first step together with the hypotheses of the theorem and the fact that we are dealing with proof outlines, are used to show that all the conditions of m-Implements (2.12) are satisfied, thus $PO(R)$ implements $(\overline{v}).PO(A)$. The formal proof follows.

Proof:

1 $(CA_R \wedge \Box I_{PO(R)}) \Rightarrow m(CA_A \wedge \Box I_{PO(A)})$

1.1 $(CA_R \wedge \Box I_{PO(R)})$	assumption
1.2 $I_{PO(R)} \Rightarrow m(I_{PO(A)})$	Lemma 3
1.3 $\Box I_{PO(R)}$	$\wedge$-elim 1.1
1.4 $\Box m(I_{PO(A)})$	TL and MP 1.3, 1.2
1.5 $m(\mathrm{DEF}^*(A))$	H-4

1.6 $m(\text{CNSB}^*(A))$ — CA_R of 1.1 and Lemma 5

1.7 $\bigwedge_{b \in \mathcal{A}_A}(\Box(m(in(b)) \Rightarrow (m(in(b)) \mathsf{U} m(after(b)))))$

1.7.1 For every $b \in \mathcal{A}_A$: $h(b) \in Comps^*(R)$ — R is a structural refinement of A

1.7.2 For every $b \in \mathcal{A}_A$: $in(b) = at(b)$ — Equation (3.3)

1.7.3 For every $b \in \mathcal{A}_A$: $m(in(b)) = m(at(b))$ — 1.7.2 and H-4

1.7.4 $\bigwedge_{b \in \mathcal{A}_A}(\Box(in(h(b)) \Rightarrow (in(h(b)) \mathsf{U} after(h(b)))))$ — 1.1, Equation (3.17)

1.7.5 $\bigwedge_{b \in \mathcal{A}_A}(\Box(m(at(b)) \Rightarrow (m(at(b)) \mathsf{U} m(after(b)))))$ — MP Lemma 4, 1.7.4

1.7.6 $\bigwedge_{b \in \mathcal{A}_A}(\Box(m(in(b)) \Rightarrow (m(in(b)) \mathsf{U} m(after(b)))))$ — 1.7.5, 1.7.3

1.8 $\Box(after(R) \Rightarrow \Box(after(R)))$ — $\wedge$-elim on 1.1

1.9 $\Box(m(after(A)) \Rightarrow \Box m((after(A))))$ — MP Lemma 6, 1.8

1.10 $m(CA_A)$ — $\wedge$-incl 1.9, 1.7, 1.6, 1.5

1.11 $m(CA_A) \wedge \Box m(I_{PO(A)})$ — $\wedge$-incl 1.10, 1.4

1.12 $m(CA_A \wedge \Box I_{PO(A)})$ — H-5 on 1.11

So, we have shown

$$(CA_R \wedge \Box I_{PO(R)}) \Rightarrow m(CA_A \wedge \Box I_{PO(A)})$$

which is condition 2c of m-Implements (2.12). Due to the hypothesis that $N^E(R) = N^E(A)$, condition 1 of m-Implements (2.12) is satisfied. The hypothesis that m is a function and that it satisfies Hypothesis H-1, implies that condition 2a of m-Implements (2.12) is satisfied. Finally, as we argued in Section 3.2, condition 2b of m-Implements (2.12) is satisfied as well. Thus, by m-Implements (2.12), we conclude that $PO(R)$ implements $(\overline{v}).PO(A)$. □

3.4 Alternative Mapping for Control Variables

In Theorem 2 (implements) we made a choice with respect to the definition of $m(at(a))$, for any action $a \in \mathcal{A}_H^I$. We decided that as long as control is inside $h(a)$ it will be as if control is at the start of a. This decision led to Hypothesis H-2 of the theorem,

$$m(at(a)) = in(h(a))$$
$$m(after(a)) = after(h(a)).$$

This decision imposes a restriction on $h(a)$—as long as a state s of the implementing program satisfies $in(h(a))$, it must satisfy $m(pre_{PO(H)}(a)$—so that $F(s)$ satisfies $at(a) \Rightarrow pre_{PO(H)}(a)$. In order to satisfy this restriction, it may be necessary to introduce auxiliary variables. These are used to record state information used in defining m.

Another alternative is to allow the mapping of some control points inside $h(a)$ to the control point denoted by $after(a)$. Rather than stuttering only $at(a)$, as our present method does, we must now stutter $after(a)$ as well. In fact, we must require that $m(after(a)) \Rightarrow (m(after(a)) \mathsf{U} after(h(a)))$ hold for every behavior of $PO(L)$. This requirement, in turn, implies that $m(post_{PO(H)}(a))$ must hold as long as $m(after(a))$ does.

Since we have found no benefit for "switching in the middle" we decided to adopt "switching at the end".

3.5 A Remark on Structural Refinement

The significance of structural refinement is that it formalizes the well known process of program derivation by step-wise refinement. Whenever we replace an action in a program by some collection of actions, the resulting program is a structural refinement of the original one. Not only does using structural refinement simplify showing that one proof outline implements another, it is what

we do anyway in refining programs. Given these observations, structural refinement, and proof-outline specifications become the cornerstones of a methodology for deriving multiple-server implementations of a service, from single-server implementations of that service. The methodology, and examples are presented in Chapter 4.

Chapter 4

Designing Distributed Implementations of Services

We now present a methodology for designing multiple-server implementations of services. The methodology is based on constructing a structural refinement of a proof outline for a single-server implementation. In effect, we view a single-server implementation of a service as an abstraction that hides details of a multiple-server implementation. For that reason, we use A (abstract) to denote a single-server implementation and R (real) to denote the multiple-server implementation.

We start by presenting a derivation of a multiple-server implementation of a mutual exclusion service. We then explain how the methods in this example can be generalized and describe a methodology that embodies the generalization. We conclude this chapter with the derivation of multiple-server implementation of an immutable-set service.

4.1 A Mutual Exclusion Service

Consider the problem of designing a mutual exclusion service for clients $c_1, \ldots, c_n$. The program of each client has a *critical section* and a *non-critical section*. We must devise a protocol that guarantees at most one client executes in its critical

$PO(A)$: $\{nxt = 0 \wedge g = 0 \wedge (I : \bigwedge_{1 \le p \le n} tkt_p < g)\}$

cobegin $\|_{1 \le p \le n}$

$PO(c_p)$: $\{tkt_p < nxt \le g \wedge I\}$

c_p: **do** *true* $\rightarrow$

$\{tkt_p < nxt \le g \wedge I\}$

NCS_p

$\{tkt_p < nxt \le g \wedge I\}$

get_p: $\langle\ g, tkt_p := g + 1, g\ \rangle$

$\{nxt \le tkt_p \wedge I \wedge (X_p : \bigwedge_{l \ne p} tkt_l \ne tkt_p)\}$

$enter_p$: $\langle$ **if** $nxt = tkt_p \rightarrow$ **skip fi** $\rangle$

$\{nxt = tkt_p \wedge I \wedge X_p\}$

CS_p

$\{nxt = tkt_p \wedge I \wedge X_p\}$

$exit_p$: $nxt := nxt + 1$

$\{tkt_p < nxt \le g \wedge I\}$

od

$\{false\}$

coend

$\{false\}$

Figure 4.1: A single-server implementation of mutual exclusion.

section at any time. Denoting the critical section for process c_p by CS_p, an outline of such a protocol was given in Figure 1.1. In that figure, clients use the two services: tickets, and guard. Therefore, we can derive a distributed mutual exclusion service by constructing multiple-server implementations of these services. Here we consider only the guard service.

We derive a multiple-server implementation of the guard service, by using a refinement mapping and constructing a structural refinement of a single-server implementation of that service.

4.1.1 A Single-Server Implementation

A proof outline including a single-server implementation of both the tickets and guard services is given in Figure 4.1. In it, the service implementation (client

stubs and server) appears in-line in client programs. This is done to avoid reasoning in terms of procedure calls. Actions and variables that become visible due to this in-line expansion are elements of the service implementation and, therefore, are internal elements of the proof outline $PO(A)$. These include variables g, nxt, and actions get_p, $enter_p$, and $exit_p$.

Proof outline $PO(A)$ can be derived using standard techniques. The assertions in $PO(A)$ must be strong enough to prove *mutual exclusion*, a safety property that can be formalized as:

$$\Box \bigwedge_{1 \leq p,q \leq n, p \neq q} \neg(in(CS_p) \wedge in(CS_q)) \tag{4.1}$$

Viewing $PO(A)$ as a specification, we prove that every behavior of $PO(A)$ satisfies (4.1) by first showing that $I_{PO(A)} \Rightarrow \bigwedge_{1 \leq p,q \leq n, p \neq q} \neg(in(CS_p) \wedge in(CS_q))$ is valid, and then, using validity of $Inv(PO(S))$ (3.8) which asserts $\Box I_{PO(A)}$, conclude that every behavior of $PO(A)$ satisfies (4.1).

The proof outline $PO(A)$ is valid. Thus every execution of A, if started at a state that satisfies $I_{PO(A)}$, is a behavior of $PO(A)$, implying that every such execution satisfies (4.1).

4.1.2 Deriving Multiple-Server Implementations

Proof outline $PO(A)$ above is in terms of single-server implementations of both the tickets and guard services. Suppose that we desire a multiple-server implementation of the guard service. Let there be k servers, each a replica of the single server that maintained nxt in A. Thus, our goal is to derive an implementation of the guard service that is based on $nxt_1, \ldots, nxt_k$ instead of being based on nxt. To achieve this goal, we design an implementation for the specification $(\overline{v}).PO(A)$, where $\overline{v}$ is a list of the elements of the set

$$\{nxt\} \bigcup_{1 \leq p \leq n} \{at(enter_p), after(enter_p), at(exit_p), after(exit_p), enter_p, exit_p\}.$$

Observe that $(\overline{v}).PO(A)$ explicitly restricts potential implementations to change only server related elements.

To design an implementation for $(\overline{v}).PO(A)$, we postulate a mapping m_{RA} that defines nxt in terms of $nxt_1, \ldots, nxt_k$ and use m_{RA} to derive a proof outline $PO(R)$ that satisfies the hypotheses of Theorem 2 (implements). This allows the conclusion that $PO(R)$ implements $(\overline{v}).PO(A)$, which, in turn, implies that $PO(R)$ satisfies (4.1). Moreover, if $PO(R)$ is valid, then we can conclude that R, when started in a state that satisfies $I_{PO(R)}$, also satisfies (4.1).

When using Theorem 2 (implements) for proving that one proof outline implements another, candidate refinement mappings are constrained by hypotheses H-1–H-6. In fact, all that is necessary in defining the refinement mapping is a mapping of internal variables that are not also control variables because H-1–H-5 define the mapping for control variables. For the single-server implementation of the guard service, there is only one such variable: nxt.

In Section 1.2 (Equation (1.1)) we presented a mapping for nxt that required keeping $nxt_1, \ldots, nxt_k$ equal. Another possible mapping for nxt is obtained by inventing a variable *hotCopy* and defining $m_{RA}(nxt)$ to equal $nxt_{hotCopy}$. The disadvantage of using this refinement mapping is that as long as *hotCopy* does not change, the service is being implemented in terms of a single server—the one indicated by *hotCopy*. And, to allow *hotCopy* to change, requires another protocol.

Yet another mapping for nxt is

$$m_{RA}(nxt) = \max_{1 \leq l \leq k}(nxt_l). \tag{4.2}$$

This mapping defines the value of nxt to be the maximum of $nxt_1, \ldots, nxt_k$. From an engineering point of view, (4.2) is attractive because it is defined in any state and does not require that updates to $nxt_1, \ldots, nxt_k$ be coordinated. Choosing a refinement mapping reflects the experience, ingenuity, and intuition of the designer; different choices can lead to implementation that differ in their

$PO(R')$: $\{\max_{1\le l\le k}(nxt_l) = 0 \wedge g = 0 \wedge (I : \bigwedge_{1\le p\le n} tkt_p < g)\}$

cobegin $\|_{1\le p\le n}$

$PO(c_p)$: $\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

c_p: **do** *true* $\rightarrow$

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

NCS_p

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

get_p: $\langle\ g, tkt_p := g+1, g\ \rangle$

$\{\max_{1\le l\le k}(nxt_l) \le tkt_p \wedge I \wedge (X_p : \bigwedge_{l\ne p} tkt_l \ne tkt_p)\}$

$enter_p$: $\langle$ **if** $nxt = tkt_p \rightarrow$ **skip fi** $\rangle$

$\{\max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p\}$

CS_p

$\{\max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p\}$

$exit_p$: $nxt := nxt + 1$

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

od

$\{false\}$

coend

$\{false\}$

Figure 4.2: Translated assertions.

performance costs.

Having selected m_{RA} as a candidate refinement mapping, our goal is to derive a proof outline $PO(R)$ that is a structural refinement of $(\overline{v}).PO(A)$, and for which the hypotheses of Theorem 2 (implements) are satisfied.

We start by reformulating the assertions in $PO(A)$ as assertions with free variables from $N(R)$, such that an assertion Q in $PO(A)$ is reformulated $m_{RA}(Q)$. Doing this guarantees that for every external control point (i.e., a control point belonging to an external action) hypothesis H-6 is satisfied. The resulting (not necessarily valid) proof outline, $PO(R')$, is shown in Figure 4.2.

In $PO(R')$, the triples for internal actions $enter_p$, $exit_p$ are not valid—the

actions modify nxt and the assertions are in terms of $nxt_1, \ldots, nxt_k$. If we view $PO(R')$ as a specification, the fact that it is not valid is not an impediment to proving that $PO(R')$ implements $(\overline{v}).PO(A)$. However, proving that $PO(R')$ implements $(\overline{v}).PO(A)$ will not allow us to conclude anything about R' itself unless $PO(R')$ is valid. So, validity of $PO(R')$ is essential for our goal of deriving a multiple-server implementation of the mutual exclusion service. Note also that restoring validity of $PO(R')$ coincides with the goal of deriving guard service implementation in terms of $nxt_1, \ldots, nxt_k$—we want to replace actions $enter_p$ and $exit_p$ by programs that use $nxt_1, \ldots, nxt_k$ instead of nxt.

In the following, $h(a)$, $h'(a)$, etc., denote candidate replacement programs for an action $a \in \mathcal{A}_{PO(A)}$. Since a is an atomic action, any statement is a structural refinement of a. Define $PO(h(a))$ to be a proof outline of $h(a)$.

Derivation of $PO(h(a))$ is driven by two requirements. First, to satisfy Hypothesis H-6 of Theorem 2, $PO(h(a))$ must satisfy

$$I_{PO(h(a))} \wedge in(h(a)) \Rightarrow pre_{PO(R')}(a). \tag{4.3}$$

Second, in order to restore validity in $PO(R')$, proof outline $PO(h(a))$ must also satisfy

$$\begin{aligned} pre_{PO(R')}(a) &\Rightarrow pre(PO(h(a))) \\ post(PO(h(a))) &\Rightarrow post_{PO(R')}(a). \end{aligned} \tag{4.4}$$

In light of this, we now present several possible structural refinements for actions $enter_p$ and $exit_p$.

Refining $enter_p$

A first approximation for the refinement of $enter_p$ is given in Figure 4.3. This proof outline is obtained by observing that the difference between the precondition and postcondition of $enter_p$ in $PO(R')$ is that the postcondition ensures that states satisfy both the precondition and $\max_{1 \le l \le k}(nxt_l) = tkt_p$.[1] The

[1] By definition, execution of $\langle$ **if** $\max_{1 \le l \le k}(nxt_l) = tkt_p \rightarrow$ **skip fi** $\rangle$ is delayed until the condition $\max_{1 \le l \le k}(nxt_l) = tkt_p$ holds.

$PO(h(enter_p))$: $\{\max_{1\le l\le k}(nxt_l) \le tkt_p \wedge I \wedge (X_p : \bigwedge_{l\neq p} tkt_l \neq tkt_p)\}$

$h(enter_p)$: $\langle$ **if** $\max_{1\le l\le k}(nxt_l) = tkt_p \rightarrow$ **skip fi** $\rangle$

$\{\max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p\}$

Figure 4.3: A first step in refining $enter_p$.

reader can verify that (4.3) and (4.4) are both satisfied by this proof outline. Thus, $PO(h(enter_p))$ is (theoretically) an acceptable refinement of $enter_p$. However, $h(enter_p)$ specifies an action that calculates the maximum of $nxt_1, \ldots, nxt_k$ atomically, which, from a practical point of view, is an expensive operation to implement. This suggests that we look further.

The atomicity requirement of $h(enter_p)$ can be relaxed. The insight leading to that relaxation is obtained by reformulating $pre(h(enter_p))$ and $post(h(enter_p))$. By definition, $\max_{1\le l\le k}(nxt_l) \le tkt_p$ in the precondition is equivalent to

$$(\bigwedge_{1\le l\le k} (nxt_l \le tkt_p)). \tag{4.5}$$

Similarly, $\max_{1\le l\le k}(nxt_l) = tkt_p$ in the postcondition is equivalent to

$$(\bigwedge_{1\le l\le k} (nxt_l \le tkt_p)) \wedge (\bigvee_{1\le l\le k} (nxt_l = tkt_p)). \tag{4.6}$$

Observe that (4.6) results from strengthening (4.5) with the conjunct

$$\bigvee_{1\le l\le k} (nxt_l = tkt_p).$$

A **proof outline** containing an **if** statement to implement this strengthening is given in **Figure** 4.4. Although the action in Figure 4.4 does not relax the atomicity requirement, it can be implemented using a **cobegin** statement as shown in Figure 4.5. This **cobegin** terminates only if $nxt_1, \ldots, nxt_k$ are incremented whenever any one of them is, and results in an unnecessary delay for $enter_p$. To avoid this unnecessary delay, we introduce a variable $done_p$ for every client c_p, that marks the server instance j for which $nxt_j = tkt_p$ holds. One option is to

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge I \wedge X_p \}$

$\langle$ **if** $\bigvee_{1 \le l \le k} (nxt_l = tkt_p)$ $\rightarrow$ **skip fi** $\rangle$

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge \bigvee_{1 \le l \le k} (nxt_l = tkt_p) \wedge I \wedge X_p \}$

Figure 4.4: A second step in refining $enter_p$.

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge I \wedge X_p \}$

cobegin $\|_{1 \le j \le k}$

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge I \wedge X_p \}$

if $nxt_j = tkt_p$ $\rightarrow$ **skip fi**

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge nxt_j = tkt_p \wedge I \wedge X_p \}$

coend

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge \bigvee_{1 \le l \le k} (nxt_l = tkt_p) \wedge I \wedge X_p \}$

Figure 4.5: A third step in refining $enter_p$.

$PO(h'(enter_p))$: $\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge I \wedge X_p \}$

$done_p := 0$

$\{ \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p) \wedge I \wedge X_p \wedge D_p \}$

cobegin $\|_{1 \le j \le k}$

$\{(L_p : \bigwedge_{1 \le l \le k} (nxt_l \le tkt_p)) \wedge I \wedge X_p \wedge D_p\}$

$\langle$ **if** $nxt_j = tkt_p \wedge done_p = 0 \rightarrow done_p := j$

$[]$ $done_p \neq 0 \rightarrow$ **skip**

fi $\rangle$

$\{L_p \wedge (E_p : \bigvee_{1 \le l \le k}(nxt_l = tkt_p)) \wedge I \wedge X_p \wedge D_p \wedge done_p \neq 0\}$

coend

$\{L_p \wedge E_p \wedge I \wedge X_p \wedge D_p \wedge done_p \neq 0\}$

Figure 4.6: A **cobegin** refinement of $enter_p$.

implement $done_p$ as a Boolean that is initialized to *false* upon entry to the **cobegin** and set to *true* once such a server instance is found. More useful, though, is to store in $done_p$ the identity of the server instance j for which $nxt_j = tkt_p$ (i.e., the server instance that actually gave permission to enter the critical section). To do this, the following is used in $h(enter_p)$:

$$D_p\text{: } 0 \le done_p \le k \wedge done_p \neq 0 \Rightarrow nxt_{done_p} = tkt_p.$$

It is straightforward to verify that $(D_p \wedge done_p \neq 0) \Rightarrow \bigvee_{1 \le l \le k}(nxt_l = tkt_p)$. The resulting **cobegin** refinement of $enter_p$ is given in Figure 4.6. The reader may verify that $PO(h'(enter_p))$ satisfies both (4.3) and (4.4), and that it is valid.

Refining $exit_p$

Action $exit_p$ of Figure 4.2 must be replaced by a program that increments $\max_{1 \le l \le k}(nxt_l)$. One method is to increment each of $nxt_1, nxt_2, \ldots, nxt_k$. This, however, is inefficient. Another method is to find a server instance j for which $nxt_j = \max_{1 \le l \le k}(nxt_l)$ holds, and increment nxt_j. This can be described by

$h(exit_p)$: Find j such that $nxt_j = \max_{1 \le l \le k}(nxt_l)$ and increment nxt_j.

$PO(h(exit_p))$: $\{nxt_{done_p} = \max_{1 \le l \le k}(nxt_l) = tkt_p \wedge I \wedge X_p \wedge D_p \wedge done_p \neq 0\}$
$nxt_{done_p} := nxt_{done_p} + 1$
$\{tkt_p < \max_{1 \le l \le k}(nxt_l) \le g \wedge I\}$

Figure 4.7: Refining $exit_p$.

However, given $PO(h'(enter_p))$ of Figure 4.6, calculating $\max_{1 \le l \le k}(nxt_l)$ and searching for the server j for which $nxt_j = \max_{1 \le l \le k}(nxt_l)$ holds is not necessary. Due to the conjuncts D_p and $done_p \neq 0$ in $post(PO(h'(enter_p)))$, we conclude that $nxt_{done_p} = \max_{1 \le l \le k}(nxt_l)$ holds, and so incrementing nxt_{done_p}, as shown in Figure 4.7, is all that is needed. It is straightforward to verify that $PO(h(exit_p))$ is valid, and that it satisfies both (4.3) and (4.4).

A complete multiple-server implementation

A complete specification $PO(R)$ of a multiple-server implementation results from replacing actions $enter_p$ and $exit_p$ of Figure 4.2 by their proof outline refinements, $PO(h'(enter_p))$ of Figure 4.6 and $PO(h(exit_p))$ of Figure 4.7. The proof outline $PO(R)$ is presented in Figure 4.8.

Proof outline $PO(R)$ of Figure 4.8 implements $(\overline{v}).PO(A)$. This follows from the fact that due to the way we constructed R from A, $PO(R)$ is a structural refinement of $(\overline{v}).PO(A)$, and from the fact that proof outlines $PO(h'(enter_p))$ and $PO(h(exit_p))$ satisfy both (4.3) and (4.4).

The multiple-server implementation of the mutual exclusion service given in $PO(R)$ has a problem. Once some server variable nxt_j is incremented in $h(exit_p)$, this variable determines the value of $\max_{1 \le l \le k}(nxt_l)$ thereafter. Thus, some of the disadvantages of a single-server implementation are retained. This problem can be solved by introducing another program, consisting completely of internal variables and actions, that repeatedly ("in the background") updates $nxt_1, \ldots, nxt_k$ to follow $\max_{1 \le l \le k}(nxt_l)$. An outline of such a program is given in Figure 4.9.

A true multiple-server implementation of the mutual exclusion service can

$PO(R)$: $\{(P: \max_{1\le l\le k}(nxt_l) = 0 \wedge g = 0) \wedge (I: \bigwedge_{1\le p\le n} tkt_p < g)\}$

cobegin $\|_{1\le p\le n}$

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

c_p: **do** *true* $\rightarrow$

NCS_p

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

get_p: $\langle\ g, tkt_p := g+1, g\ \rangle$

$\{\bigwedge_{1\le l\le k}(nxt_l \le tkt_p) \wedge I \wedge (X_p: \bigwedge_{l\ne p} tkt_l \ne tkt_p)\}$

$h'(enter_p)$: $done_p := 0$

$\{(L_p: \bigwedge_{1\le l\le k}(nxt_l \le tkt_p)) \wedge I \wedge X_p \wedge D_p\}$

cobegin $\|_{1\le j\le k}$ $\{L_p \wedge I \wedge X_p \wedge D_p\}$

$\langle$ **if** $nxt_j = tkt_p \wedge done_p = 0 \rightarrow done_p := j$

$[]$ $done_p \ne 0 \rightarrow$ **skip**

fi $\rangle$ $\{L_p \wedge (E_p: \bigvee_{1\le l\le k}(nxt_l = tkt_p)) \wedge I \wedge X_p \wedge D_p \wedge done_p \ne 0\}$

coend

$\{\max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p \wedge D_p \wedge done_p \ne 0\}$

CS_p

$\{\max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p \wedge D_p \wedge done_p \ne 0\}$

$h(exit_p)$: $\{nxt_{done_p} = \max_{1\le l\le k}(nxt_l) = tkt_p \wedge I \wedge X_p\}$

$nxt_{done_p} := nxt_{done_p} + 1$

$\{tkt_p < \max_{1\le l\le k}(nxt_l) \le g \wedge I\}$

od $\{false\}$

coend $\{false\}$

Figure 4.8: A distributed mutual exclusion.

$\{I1: (\bigwedge_{1\le i\le k} nxt_i \le \max_{1\le l\le k}(nxt_l)) \wedge (\bigvee_{1\le i\le k} nxt_i = \max_{1\le l\le k}(nxt_l))\}$

$PO(bkgrnd)$: **cobegin** $\|_{1\le r\le k}$

$\{I1\}$

do *true* $\rightarrow nxt_r := \max(nxt_r, nxt_1)$

$\{I1\}$

$[]$ *true* $\rightarrow nxt_r := \max(nxt_r, nxt_2)$

$\{I1\}$

...

$\{I1\}$

$[]$ *true* $\rightarrow nxt_r := \max(nxt_r, nxt_k)$

$\{I1\}$

od

$\{false\}$

coend

$\{false\}$

Figure 4.9: A background update of server's state (gossip).

$PO(TR)$: $\{\max_{1\le l\le k}(nxt_l) = 0 \wedge g = 0) \wedge \bigwedge_{1\le p\le n} tkt_p < g\}$

cobegin

$PO(R)$

$\|$

$PO(bkgrnd)$

coend

$\{false\}$

Figure 4.10: A multiple-server implementation of the mutual exclusion service.

be constructed by a concurrent composition of $PO(R)$ and $PO(bkgrnd)$, resulting in $PO(TR)$ of Figure 4.10. The proof that $PO(TR)$ implements $PO(R)$ is straightforward. It is based on classifying all variables and actions of $PO(R)$ as externals, and classifying all variables and actions in $PO(bkgrnd)$ as internals. With this classification, the actual proof of $PO(TR)$ implements $PO(R)$ is based on (3.9). In [LL86], the actions of programs like $PO(bkgrnd)$ of Figure 4.9 are described as "gossip".

Finally, using the cobegin rule of [Sch], it is straightforward to show that

$PO(TR)$ is valid. This allows concluding that executions of TR, when started in a state that satisfies $I_{PO(TR)}$, are all behaviors of $PO(TR)$ and, thus, satisfy (4.1).

4.2 A Methodology for Designing Distributed Services

The derivation of the distributed mutual-exclusion algorithm in Section 4.1, illustrates a general methodology for designing distributed services. The methodology involves three steps:

1. Design a single-server implementation of the service with a proof outline $PO(A)$. Let $\overline{v}$ be the list of server variables and actions and $(\overline{v}).PO(A)$ be the resulting specification. Internal actions of $(\overline{v}).PO(A)$ are assumed to be atomic.

2. Fix a concrete state space Σ_R (i.e., $N^E(R), N^I(R)$), and select m_{RA}, a candidate refinement mapping. Define m_{RA} so that hypotheses H-1–H-5 of Theorem 2 (implements) are satisfied.

3. Use m_{RA} to obtain $PO(R)$—the multiple-server implementation—as follows:

   (a) Translate every assertion Q in $PO(A)$ to the expression $m_{RA}(Q)$. Let $PO(R')$ denote the resulting (not necessarily valid) proof outline. Note, due to this translation, for every $T \in Comps^*(R')$, the expressions

   $$pre_{PO(R')}(T) = m_{RA}(pre_{PO(A)}(T))$$
   $$post_{PO(R')}(T) = m_{RA}(post_{PO(A)}(T))$$

   both hold.

(b) Replace in $PO(R')$ every internal action a by a proof outline $PO(h(a))$ that satisfies

$$pre_{PO(R')}(a) \Rightarrow pre(PO(h(a)))$$
$$post(PO(h(a))) \Rightarrow post_{PO(R')}(a) \tag{4.7}$$

and also satisfies

$$(in(h(a)) \wedge I_{PO(h(a))}) \Rightarrow pre_{PO(R')}(h(a)). \tag{4.8}$$

(c) Ensure that for every state s in every behavior of the resulting proof outline $PO(R)$, and for every name $x \in N(A)$, the value $s(m_{RA}(x))$ is unique. □

This methodology guarantees that $PO(R)$ implements $(\overline{v}).PO(A)$, as shown by the following theorem.

Theorem 7 *If $PO(R)$ is derived from $(\overline{v}).PO(A)$ by using the methodology above, then $PO(R)$ implements $(\overline{v}).PO(A)$.*

Proof: The proof is based on showing that the hypotheses of Theorem 2 (implements) are satisfied.

First, we show that $PO(R)$ is a structural refinement of $(\overline{v}).PO(A)$. It suffices to show that (i) R is a structural refinement of A and (ii) that for every external action a of A, its structural refinement $h(a)$ is a itself (see Structural refinement of proof outlines (3.11)). To show (i), recall that any statement is a structural refinement of an atomic action. Since server invocations in A are atomic (due to step 1) and, by our methodology, R is obtained from A by replacing server invocations by statements and leaving all the rest unchanged, R is a structural refinement of A. Requirement (ii) follows from the fact that only the internal actions of A—server invocations—are changed in order to obtain R, thus, for any external action a, $h(a) = a$. Using Structural refinement of proof outlines (3.11), from (i) and (ii) we conclude that $PO(R)$ is a structural refinement of $(\overline{v}).PO(A)$.

Second, we must show that refinement mapping m_{RA} is well-defined. Since in step (3) of our methodology $PO(R)$ was derived from $PO(A)$ using a refinement mapping, and the mapping is uniquely defined for every state of $PO(R)$ and every variable name of $PO(A)$ (step 3c of the methodology), this follows trivially.

Third, we must show that hypotheses H-1 through H-6 of Theorem 2 (implements) hold. By step 2 of our methodology, hypotheses H-1 through H-5 are satisfied. We now show that hypothesis H-6 is also satisfied.

1 Consider any $a \in \mathcal{A}_A$

2 $(I_{PO(R)} \wedge m_{RA}(at(a))) \Rightarrow m_{RA}(pre_{PO(A)}(a))$

2.1 $(I_{PO(R)} \wedge m_{RA}(at(a)))$ assumption

2.2 $a \in \mathcal{A}_A^E \vee a \in \mathcal{A}_A^I$ 1

2.3 $a \in \mathcal{A}_A^E \Rightarrow m_{RA}(pre_{PO(A)}(a))$

2.3.1 $a \in \mathcal{A}_A^E$ assumption

2.3.2 $m_{RA}(pre_{PO(A)}(a))$

2.3.2.1 $at(a) \in N^E(A)$ 2.3.1

2.3.2.2 $m_{RA}(at(a)) = at(a)$ H-1, 2.3.2.1

2.3.2.3 $I_{PO(R)} \wedge at(a)$ $\wedge$-incl on 2.3.2.2, 2.1

2.3.2.4 $pre_{PO(R)}(a)$ 2.3.2.3

2.3.2.5 $m_{RA}(pre_{PO(A)}(a))$ 2.3.2.4, step 3a of the methodology.

2.3.3 $a \in \mathcal{A}_A^E \Rightarrow m_{RA}(pre_{PO(A)}(a))$ 2.3.1, 2.3.2

2.4 $a \in \mathcal{A}_A^I \Rightarrow m_{RA}(pre_{PO(A)}(a))$

2.4.1 $a \in \mathcal{A}_A^I$ assumption

2.4.2 $m_{RA}(pre_{PO(A)}(a))$

2.4.2.1 $at(a) \in N^I(A)$ 2.4.1

2.4.2.2 $m_{RA}(at(a)) = in(h(a))$ 2.4.2.1, H-2

2.4.2.3 $I_{PO(R)} \wedge in(h(a))$ — $\wedge$-incl 2.4.2.2, 2.1

2.4.2.4 $I_{PO(R)} \Rightarrow I_{PO(h(a))}$ — step 3b of the methodology.

2.4.2.5 $I_{PO(h(a))}$ — MP 2.1, 2.4.2.4

2.4.2.6 $I_{PO(h(a))} \wedge in(h(a))$ — $\wedge$-incl 2.4.2.3, 2.4.2.5

2.4.2.7 $pre_{PO(R')}(h(a))$ — MP Equation 4.8, 2.4.2.6

2.4.2.8 $m_{RA}(pre_{PO(A)}(a))$ — step 3a of the methodology, 2.4.2.7

2.4.3 $a \in \mathcal{A}_A^I \Rightarrow m_{RA}(pre_{PO(A)}(a))$ — 2.4.1, 2.4.2

2.5 $m_{RA}(pre_{PO(A)}(a))$ — $\vee$-elim 2.2, 2.3, 2.4

3 $\bigwedge_{a \in \mathcal{A}_A}((I_{PO(R)} \wedge m_{RA}(at(a))) \Rightarrow m_{RA}(pre_{PO(A)}(a)))$ — 1, 2

4 $(I_{PO(R)} \wedge m_{RA}(after(A))) \Rightarrow m_{RA}(post_{PO(A)}(A))$

4.1 $I_{PO(R)} \wedge m_{RA}(after(A))$ — assumption

4.2 $m_{RA}(post_{PO(A)}(A))$

4.2.1 $m_{RA}(after(A))$ — $\wedge$-elim 4.1

4.2.2 $after(R)$ — see Lemma 6

4.2.3 $post_{PO(R)}(R)$ — MP 4.1, 4.2.2

4.2.4 $m_{RA}(post_{PO(A)}(A))$ — 4.2.3, and 3a of the methodology.

4.3 $(I_{PO(R)} \wedge m_{RA}(after(A))) \Rightarrow m_{RA}(post_{PO(A)}(A))$ — 4.1, 4.2

5 **By 3 and 4 conclude**

$$\bigwedge_{a \in \mathcal{A}_A}((I_{PO(R)} \wedge m_{RA}(at(a))) \Rightarrow m_{RA}(pre_{PO(A)}(a)))$$
$$I_{PO(R)} \wedge m_{RA}(after(A)) \Rightarrow m_{RA}(post_{PO(A)}(A))$$

□

By ensuring that $PO(R)$ is valid, we can also conclude that executions of the resulting program R, when started in a state satisfying a given initial condition,

are all behaviors of $(\overline{v}).PO(A)$. In fact, we conjecture that if $PO(A)$ is valid, $PO(R)$ is a structural refinement of $PO(A)$, and the proof outlines replacing the internal actions of $(\overline{v}).PO(A)$ are valid and mutually interference-free, then $PO(R)$ will be valid.

4.2.1 Performance

A program R that results from applying our methodology is guaranteed to satisfy the specification, but is not guaranteed to be efficient. In general, the coarser the atomicity of operations, and the larger the number of servers that must be involved in performing an operation, the worse will be the performance of the multiple-server implementation. Two factors contribute significantly to the performance of a multiple-server implementation that is obtained using our methodology. The first is the choice of refinement mapping. In Chapter 1, we used

$$nxt = \begin{cases} nxt_1 & \text{if } true \\ \ldots & \ldots \\ nxt_n & \text{if } true \end{cases}$$

as a refinement mapping. The resulting implementation (see Figure 1.5) involved updating all servers atomically. Implementations for the same problem that were presented in this chapter, are all based on a different refinement mapping

$$nxt = \max_{1 \le l \le k}(nxt_l).$$

This refinement mapping did not impose such an atomicity requirement. So, performance issues can be addressed by considering different refinement mappings.

The second factor affecting performance is the original specification $(\overline{v}).PO(A)$ itself. The weaker the specification, the less it constrains an implementation. Sometimes, in designing a single-server implementation, one does not realize that certain assertions in $PO(A)$ are stronger than necessary. However, when using our methodology for deriving the multiple-server implementation, such strong assertions can lead to an implementation that requires large atomic actions. When

this happens, one should refer back to $PO(A)$ and check whether it is possible to weaken assertions that seem to induce the observed performance problems. We illustrate these ideas about performance in the next section, where we derive a distributed implementation of a set service and show how a change in the specification can have a dramatic effect on an implementation.

4.3 A Distributed Set

Desired is a multiple-server implementation of a monotonic-set service that supports operations *insert* and *member*. We start by specifying a single-server implementation that maintains a set S. Then, we derive an implementation of S that uses a collection of k server instances where server-instance j maintains set-instance S_j.

4.3.1 A Single-Server Implementation

The server maintains a set S and provides two operations, which can be specified as proof outlines:

$$\begin{array}{l} \{P^S_{S\cup\{x\}}\}\ \ insert(x)\ \ \{P\} \\ \{true\}\ \ member(x,y)\ \ \{y = x \in S\} \end{array} \tag{4.9}$$

4.3.2 A Multiple-Server Implementation

To derive a multiple-server implementation of the set service, we define a refinement mapping that defines set S in terms of set instances $S_1, \ldots, S_k$. Several different mappings are possible, each producing a different multiple-server implementation. We use m, m', etc., to denote the mappings.

One such mapping is

$$m(S) = \bigcup_{1\le l\le k} S_l. \tag{4.10}$$

It is fairly easy to see that m induces an inexpensive update—an element that is inserted in *any* one of the set instances is considered to be in the set. However,

this mapping does result in an expensive membership operation. In order to test for membership, all set instances might have to be tested.

Another possible mapping is

$$m'(S) = \bigcap_{1 \le l \le k} S_l. \tag{4.11}$$

In contrast to mapping m (4.10), m' causes updates to be expensive—an element must be inserted in all set instances before it is considered to be in the set. The test for membership when m' is used, also is expensive. In the worst case (when the tested element is a member of the set) all set instances must be tested.

In the following sections we derive a multiple-server implementation of the set service that is driven by m (4.10).

Implementing *insert*

We start by translating the assertions in the specification of $insert(x)$. For an assertion P, $m(P) = P^S_{\cup_{1 \le l \le k} S_l}$. It is straightforward to verify that

$$m(P^S_{S \cup \{x\}}) = (P^S_{S \cup \{x\}})^S_{\cup_{1 \le l \le k} S_l} = (P^S_{\cup_{1 \le l \le k} S_l})^{\cup_{1 \le l \le k} S_l}_{\cup_{1 \le l \le k} S_l \cup \{x\}}.$$

For notational convenience, let Q denote $P^S_{\cup_{1 \le l \le k} S_l}$, and thus the first outline is

$$\{Q^{\cup_{1 \le l \le k} S_l}_{\cup_{1 \le l \le k} S_l \cup \{x\}}\} \ insert(x) \ \{Q\}. \tag{4.12}$$

In order to make (4.12) valid, we must replace the action $insert(x)$ by actions that invoke $insert_j$ operations of server instances $1, \ldots, k$. Since S is union of S_j's, and assuming that the semantics of an $insert_j$ operation is given by

$$\{P^{S_j}_{S_j \cup \{x\}}\} \ insert_j(x) \ \{P\},$$

invoking a single $insert_j$ will suffice. Formally, since for any j, S_j appears in Q only as part of $\cup_{1 \le l \le k} S_l$, and since set union is associative, we have

$$Q^{\cup_{1 \le l \le k} S_l}_{\cup_{1 \le l \le k} S_l \cup \{x\}} = Q^{S_j}_{S_j \cup \{x\}}.$$

This observation leads to the program in Figure 4.11, which is a refinement of $insert$ that nondeterministically selects a server instance j and invokes $insert_j$.

$\{Q^{\cup_{1\le l\le k} S_l}_{\cup_{1\le l\le k} S_l \cup \{x\}}\}$
if $[]_{1\le j\le k}$ *true* $\rightarrow$
$\quad \{Q^{S_j}_{S_j \cup \{x\}}\}$
$\quad insert_j(x)$
$\quad \{Q\}$
fi
$\{Q\}$

Figure 4.11: A multiple-server implementation of *insert*.

Implementing *member*

Translating the assertions in the specification of *member* (4.9) we get the triple

$$\{true\}\ \ member(x,y)\ \ \{y = x \in \cup_{1\le l\le k} S_l\}. \tag{4.13}$$

Again, for validity we must replace $member(x,y)$ in (4.13) by operations on the server instances. Assume that each server j provides an operation $member_j$ with the following semantics

$$\{true\}\ \ member_j(x,y)\ \ \{y = x \in S_j\}. \tag{4.14}$$

One possible approach for refining *member* is to use Boolean variables $y_1, \ldots, y_k$ to collect the results from the servers and compute y as a function of the y_j's. Although correct, this is expensive. Once any of the y_j's becomes *true*, $x \in \cup_{1\le l\le k} S_l$ holds. Still, in order to assert $x \notin \cup_{1\le l\le k} S_l$, the values of all y_j's are needed. The program segment in Figure 4.12 expresses this idea.

A weaker specification for *member*

Suppose the specification of *member* is weakened to be

$$\{true\}\ \ member(x,y)\ \ \{y \Rightarrow x \in S\}. \tag{4.15}$$

To obtain a multiple-server implementation we use m to translate the assertions in (4.15). We get the proof outline

$$\{true\}\ \ member(x,y)\ \ \{y \Rightarrow x \in \cup_{1\le l\le k} S_l\}. \tag{4.16}$$

$\{true\}$
$y_1, \ldots, y_k := false, \ldots, false$
cobegin $\|_{1 \le j \le k}$
  $\{\neg y_j\}$
  if $\neg(\bigvee_{1 \le l \ne j \le k} y_l) \rightarrow$
    $\{\neg y_j\}$
    $member_j(x, y_j)$
    $\{y_j = x \in S_j\}$
  $[\!]$ $\bigvee_{1 \le l \ne j \le k} y_l \rightarrow$
    $\{\neg y_j \wedge \bigvee_{1 \le l \ne j \le k} y_l\}$
    skip
    $\{\neg y_j \wedge \bigvee_{1 \le l \ne j \le k} y_l\}$
  fi
  $\{y_j = x \in S_j \vee (\neg y_j \wedge \bigvee_{1 \le l \ne j \le k} y_l)\}$
coend
$\{\bigwedge_{1 \le j \le k} (y_j = x \in S_j \vee (\neg y_j \wedge \bigvee_{1 \le l \ne j \le k} y_l))\}$
$\{(\bigvee_{1 \le l \le k} y_l) = x \in \cup_{1 \le l \le k} S_l\}$
$y := \bigvee_{1 \le l \le k} y_l$
$\{y = x \in \cup_{1 \le l \le k} S_l\}$

Figure 4.12: A multiple-server implementation of *member* for $y = x \in S$.

$\{true\}$
if $[]_{1 \leq j \leq k}$ $true \rightarrow$
  $\{true\}$
  $member_j$(x,y)
  $\{y \Rightarrow x \in \cup_{1 \leq l \leq k} S_l\}$
fi
$\{y \Rightarrow x \in \cup_{1 \leq l \leq k} S_l\}$

Figure 4.13: A multiple-server implementation of *member* for $y \Rightarrow x \in S$.

In order to make (4.16) valid, we must replace the action *member* by some composition of actions $member_j$. To design this replacement, we assume that the semantics of $member_j$ is the one given in (4.14) above. It is straightforward to observe that the proof outline

$\{true\}$
$member_j(x, y)$
$\{y = x \in S_j\}$
$\{y \Rightarrow x \in \cup_{1 \leq l \leq k} S_l\}$

is valid. This proof outline leads to the multiple-server implementation of *member* that is given in Figure 4.13.

Comparing the program in Figure 4.12 with the one in Figure 4.13 illustrates the effect that a specification can have on the efficiency of its implementations. A designer will often use a strong requirement, such as $y = x \in S$, simply because a single server happens to provide it, although this strong requirement is not really used. When implementing a specification in some environment where the implementation costs for the strong and weak specifications differ (e.g., multiple-server implementations), use of the weak specification usually has advantages.

Another implementation of *member* could set y to *false*. Although this implementation is obviously not desirable, the only formal way of preventing such an implementation from consideration is by specifying properties involving liveness as well as safety.

Finally, the reader might have noticed that the derivation of the multiple-server implementation of the monotonic-set service was done without a concrete context of clients. In Section 4.1, where we developed a multiple-server implementation of a mutual exclusion service, the client context was explicit. The client context is necessary for arguing validity. Without it, one does not have the necessary information for proving interference freedom. In other words, proving validity of $PO(h(a))$, for some internal action a, is not enough to establish validity of the proof outline containing $PO(h(a))$. One must prove that assertions in $PO(h(a))$ are not interfered with (by other actions in the program where $PO(h(a))$ is inserted) and that actions of $PO(h(a))$ do not interfere with assertions outside $PO(h(a))$. In general, this can be done only when the context—i.e., the clients programs—is given explicitly.

Chapter 5

Conclusion

In this thesis, we have presented a new approach for designing distributed services. We consider a multiple-server implementation of a service to be correct if it implements a single-server based specification of the service. In presenting such a specification, one must distinguish the client code, which may not be changed in different implementations, from the code of client stubs and the server, which may be changed by implementations. This distinction is achieved by classifying the server and stubs as internal elements of the specification and classifying the other parts as external elements of the specification. Multiple-server implementations are derived from a specification by rewriting the client stubs and server programs, using several servers instead of one. We developed the theory of proof outlines as specifications and the notion of structural refinement, to support this derivation process.

We presented a methodology for deriving multiple-server implementations of services from single-server based specifications of the services. Thus, our methodology decomposes the problem of designing a distributed service into two phases: a single-server phase, during which a single-server implementation of the service is designed, and a multiple-server phase, during which a multiple-server implementation of the single-server based specification is developed.

One advantage of decomposing the design process in this manner is the separation of concerns that is afforded. In the first phase, one addresses only the problems inherent in implementing the service; and in the second phase one addresses only the issues of associated with a distributed implementation. Another advantage of our two-phased decomposition is that a designer can trace problems in a multiple-server implementation (correctness as well as performance) to the single-server design or to the refinement mapping or to choices made in implementing internal server actions. Without separating the design into two phases, it is not clear what design decisions are made due to inherent characteristics of the clients/service problem and what design decisions are due to the strategy adopted to coordinate servers. For example, when using a network that does not deliver requests in the order they are issued by the clients, certain ordering constraints on clients requests may have to be enforced—even when the service is implemented by a single server. Thus, ordering constraints required by a multiple-server implementation of a service are necessarily the conjunction of ordering requirements inherent in the service and ordering requirements needed for server coordination.

Another benefit of our approach over other approaches for implementing distributed services is that it allows use of different servers in implementing a service—we only require that the semantics of individual servers will allow implementing the single-server abstraction. We do not require that individual servers be exact replicas of each other.

Finally, we should point out that viewing proof outlines as specifications and the concept of structural refinement are not limited to the problem of designing multiple-server implementations of services. These concepts may be used in any design method that is based on step-wise refinement.

5.1 Relation to Other Work

The idea that a data abstraction and its implementations are related by a mapping between their state spaces, can be traced to Hoare's paper [Hoa72] on implementing data abstractions.

Dijkstra uses the notion of *abstract variables* in deriving programs (see [Dij76], Chapter 8). There, once a solution in terms of some set of abstract variables is derived and proved correct, it is refined by expressing the abstract variables in terms of some other, less abstract ones. Such a step requires replacing statements that use the abstract variables by other statements that manipulate the new set of less abstract variables.

Gries and Prins [GP85] introduced the notion of a *representation invariant*—a formula relating implementation states to abstract states. A representation invariant is used in deriving an implementation for abstract operations from their specification. The work of Feijen, van Gasteren, and Gries [FvGG87] also uses representation invariants. In [Pri87] the term *coupling invariant*, which was suggested by Feijen, is used instead. An important difference between [Hoa72] and [GP85] is that a representation invariant characterizes a relation between states of the implementation and states of the abstraction, whereas in [Hoa72] the relation is required to be functional. It seems that by adding auxiliary variables to the implementation state space one can always make this relation functional.

The common theme of the works mentioned above is that in developing a program to satisfy a given (input/output) specification, one first derives an abstract program that provably satisfies the specification, and then, by using a mapping that expresses the abstract variables in terms of concrete variables,[1] one refines the abstract program to get a concrete one that satisfies the same specification. Our work can be viewed as applying these ideas in the context of concurrent pro-

[1] Abstract and concrete are relative. Even assembly language notation is quite abstract when one considers the hardware level with its transistors, wires, and clock signals.

grams, where specifications characterize sequences of states (or actions) rather then just pairs of states. Our work however, is explicit in specifying the abstract (we use the term *internal*) and concrete (*external*) elements of the state space, and restricts transformations to the internal elements only. Also, since our work allows for explicitly defining certain actions as internal (even if all variables accessed by those action are external), it supports transformation even in the action space alone.

The work reported in this thesis was not done as an extension of the ideas in [Dij76,GP85,Hoa72], but rather as an attempt of using the notions of specification and implementation together with proof outlines to aid in developing multiple-server based distributed algorithms. The relationship to the works cited above only became apparent much later. We believe that exposing this relationship is one of the contributions of our work.

Carrol Morgan [Mor88] also tries to unify the notions of specification and implementation (i.e., program). In Morgan's work, a pair of predicates $[p, q]$ defines a *specification* statement, and any program P for which the triple $\{p\}P\{q\}$ is valid, satisfies (or implements) $[p, q]$. In our terminology, a specification statement $[p, q]$ corresponds to a triple

$$\{p\}\langle a\rangle\{q\}$$

where a has been classified as an *internal* action. An obvious and major difference between Morgan's work and ours is that Morgan addresses sequential programs, whereas we address concurrent programs.

In *Unity* [CM88], the notion of design by refinement is forcefully advocated. A key idea in the *Unity* approach is that one first designs a solution to a problem at a high level of abstraction and later maps this solution to various architectures. In principle, the mapped versions of the original solution all solve the problem that the original one did. However, the derivation of the mapped versions is not driven by a refinement mapping (or some other formal kind of mapping). Other

aspects of program transformation that are defined in *Unity* are the notions of composing programs by *union* and by *superposition*. These transformations have the property that the transformed program satisfies any property that the original one did. However, these transformations do not provide an easy (or straightforward) method for replacing abstract (internal) elements of the original program. Using an **always** section, one might express relationship between abstract and concrete variables but the *union* and *superposition* transformations provide no way of replacing abstract actions (those that manipulate the abstract variables) by concrete ones. It seems that the only way to construct a concrete program is by starting with a program in equational form (i.e., no **assign** section) and then superposing it with some set of concrete actions. Also, in many cases, it seems that the original specification must be changed (or augmented) to account for the details of the superposed program.

Another significant approach to reasoning at several levels of abstraction is the one based on *I/O Automata* [LT87,LM86]. Components of a system can be specified as I/O automata, and several automata can be composed to form another automaton. Interaction between individual automata is restricted to the sharing of input/output actions, input actions must be always enabled, and the set of actions is partitioned into external/internal ones—supporting the notion of external and internal elements of a specification. The model supports a form of liveness termed "fair executions". The emphasis of the I/O automata model, however, is not on program derivations, but on rigorous *a posteriori* verification.

On the more practical side of distributed computing, we should point out that the body of work on the state machine approach on one hand and on managing replication on the other (in particular [LL86]), strongly motivated our work. We conjectured that although the algorithms described under state machine approach look different from the ones described under, say, the Highly Ava..ble Distributed Services methodology [LL86], they are fundamentally the same. More

specifically, we conjectured that the bond that ties these seemingly different algorithms together is implementing a single-server abstraction. The state machine approach is a generic method for developing replicated servers that implement a single-server abstraction, whereas the other replication management schemes are application dependent methods to do exactly the same. We hope that our work establishes the validity of this conjecture and provides a unifying framework for designing distributed clients/service algorithms.

Another branch of research in the area of managing replication that we like to address is the work of Frank Schmuck in the context of the ISIS project at Cornell [Sch88]. This work investigates the effect of ordering constraints on the kind of server/clients problem that can be solved in a distributed system. In contrast to [Sch88], the methodology presented here addresses ordering as well as issues of degree of multicasting (i.e., how many server instances should be addressed for a certain operation). Thus, in our methodology, trade offs between multicast degree and order can be exploited. In addition, the methodology presented in this thesis does not restrict servers to be replicas of each other.

5.2 Future Work

This thesis provides a theory for reasoning on distributed algorithms of the clients/service type. It also presents a methodology for designing multiple-server implementations of services that is based on this theory. This work can, and should, be extended in several directions:

- More experience with using the theory and the methodology is needed.
- Support liveness properties.
- Support fault-tolerant implementations.

A more detailed discussion of these points follows.

5.2.1 Experience

At the initial stages of the research that lead to this thesis, we have developed a distributed deadlock detection algorithm. This was done in the spirit of our methodology but certainly not by the letter, and we would like to examine and probably redevelop the algorithm.

Another problem we would like to address is the design of a multiple-server implementation of a full-scale set service. The development in Chapter 4 serves only to demonstrate some basic concepts. We would like to include a *delete* operation and explore other options for refinement mappings.

Recently, we started investigating the problem of refining distributed algorithms that tolerate benign type of failures, such as crash failures, to algorithms that tolerate more severe type of failures, such as omission or even arbitrary failures. The problem and solutions were presented in [NT88]. Concepts developed in this thesis can be used in solving this problem, and, based on some preliminary work, we believe that this might reveal new solutions.

5.2.2 Liveness

Liveness properties are sets of behaviors where every finite prefix of a behavior can be extended such that the resulting behavior is in the property. In more intuitive terms, a liveness property characterizes executions in which something "good" actually happens. A formal discussion of safety and liveness can be found in [AS85].

The work in this thesis addresses only safety properties and should be extended to address liveness as well. Without liveness, one can always derive trivial implementations—satisfy the safety requirement simply by doing nothing. Technically, liveness is added to a specification by augmenting the behavior axioms with liveness axioms. Thus, the principles presented in Chapter 2 hold for liveness. The problem is how a theorem like Theorem 2 (implements) should be

changed, if at all, so that liveness will be addressed.

5.2.3 Failures

The issue of handling faulty servers was not addressed in this thesis. Our conjecture is that it can be addressed by designing *fault-tolerant refinement mappings*. Such a mapping is robust and well-defined even if a certain number of the server instances are faulty. If we consider the set example of Chapter 4, the refinement mappings in (4.10) and (4.11) are defined in terms of the states of all server instances and thus become undefined whenever a single server fails. A mapping that is based on, say, a majority of the servers is more robust.

Note, by reducing the number of server instances needed for the refinement mapping to be defined, one increases the redundancy of information stored and thus increases the degree of fault-tolerance. However, the requirement that the mapping must be well-defined might have an adverse effect on performance because updates to this redundant information must be coordinated.

Bibliography

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, August 1988.

[AS85] B. Alpren and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Company, 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., 1976.

[FvGG87] W. Feijen, A. van Gasteren, and D. Gries. In-situ inversion of a cyclic permutation. *Information Processing Letters*, 24(1):11–17, January 1987.

[GHW85] J. Guttag, J. Horning, and J. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation, Systems Research Center, 1985.

[Gif79] D. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, California, December 1979. ACM SIGOPS.

[GP85] D. Gries and J. Prins. A new notion of encapsulation. In *Proceedings Sigplan 85 Symposium on Language Issues in Programming Environments*, pages 131–139. SIGPLAN, 1985.

[Hoa72] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, (1):271–281, 1972. Also in, Programming Methodology, A collection of articles by members of IFIP WG2.3, Edited by David Gries, Springer Verlag, New York 1978.

[HPSS87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LL86] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *ACM Symposium on Principles of Distributed Computing*, pages 29–39, 1986.

[LM86] N. A. Lynch and M. Merrit. Introduction to the theory of nested transactions. Technical Report 367, Massachusetts Institute of Technology, 1986.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's dissertation, Massachusetts Institute of Technology, April 1987. Also, Technical Report: mit/lcs/tr-387.

[Mor88] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. Boyer and J. Moore, editors, *The correctness problem in Computer Science*, pages 215–273. Academic Press, 1981.

[NT88] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 248–262, Toronto, Ontario, August 1988. ACM SIGOPS-SIGACT.

[Pri87] J. F. Prins. *Partial Implementations in Program Derivation.* Ph.D. dissertation, Department of Computer Science, Cornell University, 1987.

[Sch] F. B. Schneider. On concurrent programming. To appear. Also as class notes for CS613, Spring 1989, Cornell University.

[Sch86] F. B. Schneider. The state machine approach: a tutorial. Technical Report 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

[Sch88] F. B. Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* Ph.D. dissertation, Department of Computer Science, Cornell University, 1988.

[Spi89] J. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

END